MPI: A Message-Passing Interface Standard Version 3.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

July 12, 2012

ticket0.

ticket 0. 1	This document describes the Message-Passing Interface (MPI) standard, version [2.2]3.0.
2	The MPI standard includes point-to-point message-passing, collective communications, group
3	and communicator concepts, process topologies, environmental management, process cre-
4	ation and management, one-sided communications, extended collective operations, external
5	interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for
ticket0. ⁶	C, C++ and Fortran are defined.
7	[Technically, this version of the standard is based on "MPI: A Message-Passing Interface
8	Standard, version 2.1, June 23, 2008. The MPI Forum added seven new routines and a
9	number of enhancements and clarifications to the standard.]
10	Historically, the evolution of the standards is from MPI-1.0 (June 1994) to MPI-1.1
11	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
12	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality,
13	to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
14	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
ticket0. ¹⁵	combining the previous documents. [This version, MPI-2.2, is based on MPI-2.1 and provides
16	additional clarifications and errata corrections as well as a few enhancements. Version MPI-
17	
18	2.2 (September 2009) added additional clarifications and seven new routines. This version,
19	MPI-3.0, is an extension of MPI-2.2.
20	
20	
22	
22	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
ticket 0.45	©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2010 University of Tennessee, Knoxville,
46	Tennessee. Permission to copy without fee all or part of this material is granted, provided
47	the University of Tennessee copyright notice and the title of this document appear, and
48	notice is given that copying is by permission of the University of Tennessee.

Unofficial Draft for Comment Only

 Version 3.0: xx, x, 2011. Coincident with the development of MPI-2.2, the MPI Forum began discussions of a major extension to MPI. This document contains the MPI-3 Standard. This draft version of the MPI-3 standard extends the collective operations by including nonblocking versions. Unlike MPI-2.2, this standard is considered a major update to the MPI standard. As with previous versions, new features have been adopted only when there were compelling needs for the users. Some features, however, may have more than a minor impact on existing MPI implementations. Version 2.2: September 4, 2009. This document contains mostly corrections and clarifications to the [MPI 2.1]MPI-2.1 document. A few extensions have been added; however all correct [MPI 2.1]MPI-2.1 programs are correct [MPI 2.2]MPI-2.2 programs. New features were adopted only when there were compelling needs for users, open source implementations, and minor impact on existing MPI implementations. Version 2.1: June 23, 2008. This document combines the previous documents MPI-1.3 (May 30, 2008) and MPI-2.0 (July 18, 1997). Certain parts of MPI-2.0, such as some sections of Chapter 4, Miscellany, and Chapter 7, Extended Collective Operations have been merged 	 ticket0. ticket0. ticket0. ticket0. ticket0. ticket0. ticket0. ticket0. ticket10. ticke10. tick
 into the Chapters of MPI-1.3. Additional errata and clarifications collected by the MPI Forum are also included in this document. Version 1.3: May 30, 2008. This document combines the previous documents MPI-1.1 (June 12, 1995) and the MPI-1.2 Chapter in MPI-2 (July 18, 1997). Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 are also included in this document. Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. A miscellany chapter discusses items that [don't]do not fit elsewhere, in particular language interoperability. 	20 21 22 23 24 25 26 27 28 29 30 31 ticket0. 32 33 24
 Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the standard ["] "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only new function in MPI-1.2 is one for identifying to which version of the MPI Standard the implementation conforms. There are small differences between MPI-1 and MPI-1.1. There are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2 and MPI-2. Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum reconvened to correct errors and to make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1[, which is this document]. The changes from Version 1.0 are minor. A version of this document with all changes marked is available. [This paragraph is an example of a change.] 	34 35 ticket0. 36 37 38 39 40 41 42 43 ticket0. 44 ticket0. 45 46 ticket0. 47 48

1	Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation
ticket0. ²	from over 40 organizations, has been meeting since January 1993 to discuss and to define a
3	set of library interface standards for message passing. MPIF is not sanctioned or supported
4	by any official standards organization.
5	The goal of the Message-Passing Interface, simply stated, is to develop a widely used
6	
ticket0. ⁷	practical, portable, efficient, and flexible standard for message-passing.
8	[This is the final report, Version 1.0, of the Message-Passing Interface Forum.]This
9	document contains all the technical features proposed for the interface. This copy of the
10	draft was processed by IAT_EX on May 5, 1994.
11	Please send comments on MPI to mpi-comments@mpi-forum.org. Your comment will
12	be forwarded to MPI Forum committee members who will attempt to respond.
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24 25	
26	
27	
28	
29	
30	
31	
32	
33	3
34	L Contraction of the second
35	
36	;
37	,
38	3
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

Contents

A	cknov	ix		
1	$1.1 \\ 1.2 \\ 1.3 \\ 1.4 \\ 1.5 \\ 1.6 \\ 1.7 \\ 1.8 \\ 1.9 \\ 1.10$	coduction to MPIOverview and GoalsBackground of MPI-1.0Background of MPI-1.1, MPI-1.2, and MPI-2.0Background of MPI-1.3 and MPI-2.1Background of MPI-2.2Background of MPI-3.0Who Should Use This Standard?What Platforms Are Targets For Implementation?What Is Included In The Standard?What Is Not Included In The Standard?Organization of this Document	$ \begin{array}{c} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \\ 5 \\ 5 \\ 5 \\ 6 \\ 6 \\ 6 \end{array} $	
2	MPI 2.1 2.2 2.3 2.4 2.5	I Terms and ConventionsDocument NotationNaming ConventionsProcedure SpecificationSemantic TermsData Types2.5.1 Opaque Objects2.5.2 Array Arguments2.5.3 State2.5.4 Named Constants2.5.5 Choice2.5.6 Addresses2.5.7 File Offsets2.5.8 CountsLanguage Binding	<pre>9 9 10 11 12 12 14 15 15 16 16 16 17 17 17</pre>	
	2.7 2.8	2.6.1 Deprecated and RemovedNames and Functions 2.6.2 Fortran Binding Issues 2.6.3 C Binding Issues 2.6.4 C++ Binding Issues 2.6.5 Functions and Macros Processes	18 18 19 20 23 23 23 23	ticket0.341.

	2.9	Impler	nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Exam	ples
_			
3			Point Communication 27
	3.1		uction $\ldots \ldots 27$
	3.2		ng Send and Receive Operations
		3.2.1	Blocking Send
		3.2.2	Message Data
		3.2.3	Message Envelope
		3.2.4	Blocking Receive
		3.2.5	Return Status
		3.2.6	Passing MPI_STATUS_IGNORE for Status
	3.3		Type Matching and Data Conversion 37
		3.3.1	Type Matching Rules 37
			Type MPI_CHARACTER 39
		3.3.2	Data Conversion 40
	3.4		unication Modes
	3.5		tics of Point-to-Point Communication
	3.6	Buffer	Allocation and Usage
		3.6.1	Model Implementation of Buffered Mode
	3.7		ocking Communication
		3.7.1	Communication Request Objects
		3.7.2	Communication Initiation
		3.7.3	Communication Completion
		3.7.4	Semantics of Nonblocking Communications
		3.7.5	Multiple Completions 63
		3.7.6	Non-destructive Test of status
	3.8	Probe	and Cancel
		3.8.1	Probe
		3.8.2	Matching Probe
		3.8.3	Matched Receives
		3.8.4	Cancel
	3.9	Persist	tent Communication Requests
	3.10	Send-I	Receive
	3.11	Null P	rocesses
	-		
4		atypes	
	4.1		d Datatypes \dots 91
		4.1.1	Type Constructors with Explicit Addresses 93
		4.1.2	Datatype Constructors 93
		4.1.3	Subarray Datatype Constructor 103
		4.1.4	Distributed Array Datatype Constructor
		4.1.5	Address and Size Functions
		4.1.6	Lower-Bound and Upper-Bound Markers 112
		4.1.7	Extent and Bounds of Datatypes
		4.1.8	True Extent of Datatypes 116

		4.1.9	Commit and Free	. 117
		4.1.10	Duplicating a Datatype	. 119
		4.1.11	Use of General Datatypes in Communication	. 119
			Correct Use of Addresses	
			Decoding a Datatype	
			Examples	
	4.2		und Unpack	
	4.3		ical MPI_PACK and MPI_UNPACK	
5	Coll	ective	Communication	151
	5.1	Introd	uction and Overview	. 151
	5.2	Comm	unicator Argument	. 154
		5.2.1	Specifics for Intracommunicator Collective Operations	. 154
		5.2.2	Applying Collective Operations to Intercommunicators	. 155
		5.2.3	Specifics for Intercommunicator Collective Operations	
	5.3	Barrie	r Synchronization	. 157
	5.4	Broade	$\operatorname{cast}^{\circ}$. 158
		5.4.1	Example using MPI_BCAST	. 159
	5.5	Gather	r	
			Examples using MPI_GATHER, MPI_GATHERV	
	5.6		r	
		5.6.1	Examples using MPI_SCATTER, MPI_SCATTERV	
	5.7	Gather	r-to-all	
		5.7.1	Example using MPI_ALLGATHER	
	5.8	All-to-	All Scatter/Gather	
	5.9		Reduction Operations	
		5.9.1	Reduce	
		5.9.2	Predefined Reduction Operations	
		5.9.3	Signed Characters and Reductions	
		5.9.4	MINLOC and MAXLOC	
		5.9.5	User-Defined Reduction Operations	
		0.0.0	Example of User-defined Reduce	
		5.9.6	All-Reduce	
			Process-[l]Local [r]Reduction	
	5 10		e-Scatter	
	0.10		MPI_REDUCE_SCATTER_BLOCK	
			MPI_REDUCE_SCATTER	
	5 11			
	0.11		$\begin{array}{cccccccccccccccccccccccccccccccccccc$	
			Exclusive Scan	
			Example using MPI_SCAN	
	5 1 2		ocking Collective Operations	
	0.14		Nonblocking Barrier Synchronization	
			Nonblocking Broadcast	
		0.14.4	Example using MPI_IBCAST	
		5 1 9 9	Nonblocking Gather	
			Nonblocking Gattler	
			Nonblocking Gather-to-all	
		0.12.0		. 211

ticket0. ticket0.

		5.12.6	Nonblocking All-to-All Scatter/Gather	219
		5.12.7	Nonblocking Reduce	223
		5.12.8	Nonblocking All-Reduce	224
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks	225
		5.12.10	Nonblocking Reduce-Scatter	226
				227
			-	228
	5.13			228
~				~~~
6				237
	6.1			237
		6.1.1		237
		6.1.2 D	**	238
	6.2		1	240
		6.2.1	*	240
		6.2.2		240
		6.2.3		241
		6.2.4		241
	6.3	-		242
		6.3.1	*	242
		6.3.2	*	244
		6.3.3		249
	6.4	Comm	unicator Management	250
		6.4.1	Communicator Accessors	250
		6.4.2		252
		6.4.3	Communicator Destructors	260
	6.5	Motiva	ating Examples	261
		6.5.1	Current Practice $\#1$	261
		6.5.2	Current Practice #2	262
		6.5.3	(Approximate) Current Practice #3	262
		6.5.4	Example #4	263
		6.5.5	Library Example #1	264
		6.5.6	Library Example $#2$	265
	6.6	Inter-O		268
		6.6.1	Inter-communicator Accessors	269
		6.6.2	Inter-communicator Operations	271
		6.6.3	Inter-Communication Examples	274
				274
				275
	6.7	Cachir	· · · ·	276
		6.7.1		277
		6.7.2		278
		6.7.3		284
		6.7.4	Datatypes	288
		6.7.5		292
		6.7.6		292
	6.8			294
	6.9		lizing the Loosely Synchronous Model	299
	0.0	1 or ma		200

		6.9.1	Basic Statements	299	
		6.9.2	Models of Execution	299	
			Static communicator allocation	299	
			Dynamic communicator allocation	300	
			The General [c]Case	300	ticket0.
7	Pro	cess T	opologies	301	
	7.1	Introd	luction	301	
	7.2	Virtua	al Topologies	302	
	7.3	Embe	dding in MPI	302	
	7.4	Overv	iew of the Functions	302	
	7.5	Topol	ogy Constructors	304	
		7.5.1	Cartesian Constructor	304	
		7.5.2	Cartesian Convenience Function: MPI_DIMS_CREATE	305	
		7.5.3	[General (Graph)]GraphConstructor	306	ticket 259.
		7.5.4	[Distributed (Graph)]Distributed GraphConstructor	308	ticket 259.
		7.5.5	Topology Inquiry Functions		
		7.5.6	Cartesian Shift Coordinates	323	
		7.5.7	Partitioning of Cartesian [s]Structures	325	ticket0.
		7.5.8	Low-Level Topology Functions		
	7.6	Neigh	borhood Collective Communication on Process Topologies		
		7.6.1	Neighborhood Gather		
		7.6.2	Neighbor Alltoall		
	7.7		ocking Neighborhood Communication on Process Topologies		
		7.7.1	Nonblocking Neighborhood Gather		
		7.7.2	Nonblocking Neighborhood Alltoall		
	7.8		pplication Example		
8	MPI	Envi	ronmental Management	349	
	8.1		mentation Information		
	0.1	8.1.1	Version Inquiries		
		8.1.2	Environmental Inquiries		
		0.1.2	Tag Values		
			Host Rank	351	
			IO Rank		
			Clock Synchronization		
			Inquire Processor Name		ticket255.
	8.2	Memo	bry Allocation		UICKC0200.
	8.3		Handling		
	0.0	8.3.1	Error Handlers for Communicators		
		8.3.2	Error Handlers for Windows		
		8.3.3	Error Handlers for Files		
		8.3.4	Freeing Errorhandlers and Retrieving Error Strings		
	Q 1		Codes and Classes		
	8.4 8.5				
	8.5 8.6		Classes, Error Codes, and Error Handlers		
	8.6	~	\mathbf{s} and Synchronization		
	8.7	Startu			
		8.7.1	Allowing User Functions at Process Termination	378	

		8.8	8.7.2 Determining Whether MPI Has Finished	
	9	The	Info Object	383
	10	Pro	cess Creation and Management	389
			Introduction	389
		10.2	The Dynamic Process Model	390
			10.2.1 Starting Processes	390
		10.0	10.2.2 The Runtime Environment	390
		10.3	Process Manager Interface	392
			10.3.1 Processes in MPI	392
			10.3.2 Starting Processes and Establishing Communication	$392 \\ 397$
			10.3.3Starting Multiple Executables and Establishing Communication10.3.4Reserved Keys	400
			10.3.5 Spawn Example	400
ticket0.			Manager-worker Example [,]Using MPI_COMM_SPAWN.	401
tieneto.		10.4	Establishing Communication	403
			10.4.1 Names, Addresses, Ports, and All That	403
			10.4.2 Server Routines	405
			10.4.3 Client Routines	407
			10.4.4 Name Publishing	408
			10.4.5 Reserved Key Values	411
			10.4.6 Client/Server Examples	411
			Simplest Example — Completely Portable	411
			Ocean/Atmosphere - Relies on Name Publishing	412
			Simple Client-Server Example.	412
		10.5	Other Functionality	413
			10.5.1 Universe Size	413
			10.5.2 Singleton MPI_INIT	414
			10.5.3 MPI_APPNUM	415
			10.5.4 Releasing Connections	$416 \\ 417$
			10.5.5 Another way to Establish MFT Communication	417
	11		-Sided Communications	421
			Introduction	421
		11.2	Initialization	$422 \\ 423$
			11.2.1 Window Creation11.2.2 Window That Allocates Memory	423 425
			11.2.3 Window That Allocates Shared Memory	423 427
			11.2.4 Window of Dynamically Attached Memory	429
ticket270.			11.2.5 Window Destruction	432
			11.2.6 Window Attributes	433
		11.3	Communication Calls	435
			11.3.1 Put	436
			11.3.2 Get	438
			11.3.3 Examples for Communication Calls	439
			11.3.4 Accumulate Functions	441

		Accumulate Function	41
		Get Accumulate Function	44
			45
			46
			48
	11.4		53
		•	54
	11.0		58
			59
		0	64
			67
			69
			70
	11.6		70
			70
			71
	11.7	Semantics and Correctness	71
		11.7.1 Atomicity	80
		11.7.2 Ordering	80
		11.7.3 Progress	81
		11.7.4 Registers and Compiler Optimizations	83
	11.8		84
		•	
12	Exte	ernal Interfaces 49	93
	12.1	Introduction	93
	12.2	Generalized Requests	93
		12.2.1 Examples	98
	12.3	Associating Information with Status	00
	12.4	MPI and Threads	02
			03
			04
			05
			00
13	I/O	53	11
	13.1	Introduction	11
		13.1.1 Definitions	11
	13.2		13
		*	13
			16
			16
			$10 \\ 17$
			18
		• • •	19
		• • •	19
			21
			22
			24
	13/	Data Access	27

	13.4.1	Data Access Routines	 	 		 527
		Positioning	 	 		 528
		Synchronism	 	 		 528
		Coordination	 	 		 529
		Data Access Conventions	 	 		 529
	13.4.2	Data Access with Explicit Offsets				530
		Data Access with Individual File Pointers .				534
		Data Access with Shared File Pointers				542
	101111	Noncollective Operations				543
		Collective Operations				545
		Seek				547
	13/15	Split Collective Data Access Routines				548
12.5		teroperability				556
10.0						$550 \\ 558$
		Datatypes for File Interoperability				
		External Data Representation: "external32"				560
	13.5.3	User-Defined Data Representations				561
		Extent Callback				563
		Datarep Conversion Functions				564
		Matching Data Representations				566
13.6		tency and Semantics				567
		File Consistency				567
		Random Access vs. Sequential Files				570
	13.6.3	Progress	 	 	 •	 570
		Collective File Operations				570
		Type Matching				571
	13.6.6	Miscellaneous Clarifications	 	 		 571
	13.6.7	MPI_Offset Type	 	 		 571
	13.6.8	Logical vs. Physical File Layout	 	 		 571
	13.6.9	File Size	 	 		 572
		Examples				572
		Asynchronous I/O	 	 		 575
13.7	I/O Er	$\operatorname{rror}^{'}$ Handling				576
		rror Classes				577
		\mathbf{bles}				577
	-	Double Buffering with Split Collective I/O				577
		Subarray Filetype Constructor				580
14 Too	l Supp	ort				584
14.1	Introd	uction	 	 	 •	 584
14.2	Profili	ng Interface	 	 		 584
	14.2.1	Requirements	 	 		 584
	14.2.2	Discussion	 	 		 585
	14.2.3	Logic of the Design	 	 		 586
	14.2.4	Miscellaneous Control of Profiling	 	 		 586
		Profiler Implementation [Example				587
		MPI Library Implementation [Example				587
		Systems with Weak Symbols				588
		Systems Without Weak Symbols				588
		v v v v v v v v v v v v v v v v v v v				

ticket266. ticket266.

	14.2.	7 Complications	589
		Multiple Counting	589
		Linker Oddities	589
		Fortran Support Methods	589
	14.2.	8 Multiple Levels of Interception	589
14		MPI Tool Information Interface	
		1 Verbosity Levels	
		2 Binding MPI Tool Information Interface Variables to MPI Objects .	
		3 Convention for Returning Strings	
		4 Initialization and Finalization	
		5 Datatype System	
		6 Control Variables	
		Control Variable Query Functions	
		Example: Printing All Control Variables	
		Handle Allocation and Deallocation	
		Control Variable Access Functions	
		Example: Reading the Value of a Control Variable	
	14.3.	7 Performance Variables	
		Performance Variable Classes	
		Performance Variable Query Functions	
		Performance Experiment Sessions	
		Handle Allocation and Deallocation	
		Starting and Stopping of Performance Variables	
		Performance Variable Access Functions	
		Example: Tool to Detect Receives with Long Unexpected Message	
		Queues	
	14.3.	8 Variable Categorization	
		9 Return Codes for the MPI tool information interface	
		10 Profiling Interface	
		ed Functions	619
15	.1 Depr	$ecated since MPI-2.0 \dots \dots$	619
	-	ecated since MPI-2.2 \ldots	
15	.3 Depr	ecated since MPI-3.0 \ldots	627
10 T			
		e Bindings	629
16			
		1 Overview	
		2 Design	
		3 C++ Classes for MPI	
		4 Class Member Functions for MPI	
		5 Semantics	
		$6 C++ Datatypes \ldots \ldots$	
		7 Communicators	
		8 Exceptions	
		9 Mixed-Language Operability	
		10 Profiling	
16	.2 Fortr	an Support	642

	16.2.1 Overview	642
ticket230-B.	16.2.2 Fortran Support Through the mpi_f08 Module	644
ticket230-B.	16.2.3 [Extended] Fortran Support Through the mpi Module	646
ticket230-B.	The mpi Module	647
		649
ticket230-B.	16.2.4 [Basic] Fortran Support Through the mpif.h Include File	649
ticket230-B.		651
ticket247-S.		656
ticket 247-S.	16.2.7 Requirements on Fortran Compilers	659
ticket 230-B.		661
ticket238-J.	16.2.9 Additional Support for Fortran Numeric Intrinsic Types	662
	Parameterized Datatypes with Specified Precision and Exponent Range	
	Support for Size-specific MPI Datatypes	667
	Communication With Size-specific Types	669
	16.2.10 Problems With Fortran Bindings for MPI	670
		672
	0 11 0	672
ticket236-H.	16.2.12 Problems Due to Data Copying and Sequence Association with Sub-	012
0101101200 11.		673
		673
ticket236-H.	16.2.13 Problems Due to Data Copying and Sequence Association with Vector	010
UCKC0200-11.		676
		676
		676
		677
		677
	U 1	679
		679
ticket238-J.	16.2.17 Problems with Code Movement and Register Optimization	680
UCKet230-J.		680
		681
		681
ticket238-J.		682
ticket238-J.		683
ticket238-J.		684
ticket250-J.	Calling MPI_F_SYNC_REG	685
	Module variables and COMMON blocks	
		686
4:-14020 I		686
ticket238-J.		686 687
ticket238-J.		687
ticket 238-J.	16.2.19 Permanent Data Movement	688 688
		688 604
		694 694
	16.3.1 Introduction $\dots \dots \dots$	694 694
	1	694 694
		694
		695 695
	16.3.5 Status \ldots	698

		16.3.6	MPI Opaque Objects	701	
			Datatypes	701	
			Callback Functions	703	
			Error Handlers	703	
			Reduce Operations	703	
			Addresses	703	
		16.3.7	Attributes	704	
		16.3.8	Extra State	708	
		16.3.9	Constants	708	
		16.3.10	Interlanguage Communication	709	
A	Lang	guage	Bindings Summary	711	
	A.1	Define	d Values and Handles	711	
		A.1.1	Defined Constants	711	
		A.1.2	Турез	727	
		A.1.3	Prototype [d] Definitions	728	ticket0.
			C Bindings	728	
			Fortran 2008 Bindings with the mpi_f08 Module	729	
			Fortran Bindings with mpif.h or the mpi Module	732	
			C++ Bindings (deprecated)	734	
		A.1.4	Deprecated [p]Prototype [d]Definitions	735	ticket0.
		A.1.5	Info Keys	735	ticket0.
		A.1.6	Info Values	736	
	A.2	C Bind	lings	737	
		A.2.1	Point-to-Point Communication C Bindings	737	
		A.2.2	Datatypes C Bindings	739	
		A.2.3	Collective Communication C Bindings	740	
		A.2.4	Groups, Contexts, Communicators, and Caching C Bindings	743	
		A.2.5	Process Topologies C Bindings	746	
		A.2.6	MPI Environmental Management C Bindings	748	
		A.2.7	The Info Object C Bindings	749	
		A.2.8	Process Creation and Management C Bindings	749	
		A.2.9	One-Sided Communications C Bindings	750	
		A.2.10	External Interfaces C Bindings	752	
		A.2.11	I/O C Bindings	752	
		A.2.12	Language Bindings C Bindings	755	
		A.2.13	Profiling Interface C Bindings	756	
		A.2.14	Deprecated C Bindings	756	
	A.3	Fortra	n 2008 Bindings with the mpi_f08 Module	757	
		A.3.1	Point-to-Point Communication Fortran 2008 Bindings	757	
		A.3.2	Datatypes Fortran 2008 Bindings	762	
		A.3.3	Collective Communication Fortran 2008 Bindings	766	
		A.3.4	Groups, Contexts, Communicators, and Caching Fortran 2008 Binding	<mark>ss</mark> 774	
		A.3.5	Process Topologies Fortran 2008 Bindings	780	
		A.3.6	MPI Environmental Management Fortran 2008 Bindings	785	
		A.3.7	The Info Object Fortran 2008 Bindings	788	
		A.3.8	Process Creation and Management Fortran 2008 Bindings	789	
		A.3.9	One-Sided Communications Fortran 2008 Bindings	790	

		A.3.10 External Interfaces Fortran 2008 Bindings	
		A.3.11 I/O Fortran 2008 Bindings	796
		A.3.12 Language Bindings Fortran 2008 Bindings	
		A.3.13 Profiling Interface Fortran 2008 Bindings	804
		A.3.14 Deprecated Fortran 2008 Bindings	804
	A.4	Fortran Bindings with mpif.h or the mpi Module	805
		A.4.1 Point-to-Point Communication Fortran Bindings	805
		A.4.2 Datatypes Fortran Bindings	808
		A.4.3 Collective Communication Fortran Bindings	810
		A.4.4 Groups, Contexts, Communicators, and Caching Fortran Bindings .	814
		A.4.5 Process Topologies Fortran Bindings	818
		A.4.6 MPI Environmental Management Fortran Bindings	820
		A.4.7 The Info Object Fortran Bindings	822
		A.4.8 Process Creation and Management Fortran Bindings	823
		A.4.9 One-Sided Communications Fortran Bindings	824
		A.4.10 External Interfaces Fortran Bindings	827
		A.4.11 I/O Fortran Bindings	
		A.4.12 Language Bindings Fortran Bindings	
		A.4.13 Profiling Interface Fortran Bindings	832
		A.4.14 Deprecated Fortran Bindings	
	A.5	C++ Bindings (deprecated)	835
		A.5.1 Point-to-Point Communication C++ Bindings	
		A.5.2 Datatypes C++ Bindings	838
		A.5.3 Collective Communication C++ Bindings	840
		A.5.4 Groups, Contexts, Communicators, and Caching C++ Bindings	843
		A.5.5 Process Topologies C++ Bindings	846
		A.5.6 MPI Environmental Management C++ Bindings	
		A.5.7 The Info Object C++ Bindings	848
		A.5.8 Process Creation and Management C++ Bindings	849
		A.5.9 One-Sided Communications C++ Bindings	
		A.5.10 External Interfaces C++ Bindings	851
		A.5.11 I/O C++ Bindings	851
		A.5.12 Language Bindings C++ Bindings	855
		A.5.13 Profiling Interface C++ Bindings	
		A.5.14 C++ Bindings on all MPI Classes	
		A.5.15 Construction / Destruction	
		A.5.16 Copy / Assignment	856
		A.5.17 Comparison	856
		A.5.18 Inter-language Operability	
-		.	070
в		inge-Log	859
	B.1	Changes from Version 2.2 to Version 3.0	
	B.2	TEST for Tickets 271, 168, 204, 280, 286, 287	
	B.3	Changes from Version 2.1 to Version 2.2	
	B.4	Changes from Version 2.0 to Version 2.1	870
Bi	bliog	graphy	875

Examples Index	880
MPI Constant and Predefined Handle Index	883
MPI Declarations Index	891
MPI Callback Function Prototype Index	893
MPI Function Index	894

List of Figures

5.1	Collective comminucations, an overview	153
5.2	Intercommunicator allgather	156
5.3		157
5.4	Gather example	164
5.5		165
5.6		166
5.7	Gathery example, 2-dimensional, subarrays with different sizes	167
5.8	Gathery example, 2-dimensional, subarrays with different sizes and strides .	169
5.9	Scatter example	174
5.10	Scattery example with strides	174
5.11	Scattery example with different strides and counts	176
5.12	Race conditions with point-to-point and collective communications	231
5.13	Overlapping Communicators Example	235
6.1	9 – –	256
6.2		259
6.3		274
6.4	Three-group ring[ticket0.][.]	275
7.1	Set-up of process structure for two-dimensional parallel Poisson solver	344
7.1		$\frac{344}{345}$
7.2	Communication routine with local data copying and sparse neighborhood	949
1.5		346
7.4	Communication routine with sparse neighborhood all-to-all-w and without	340
1.4		347
	local data copying.	347
11.1	Schematic description of [ticket270.][window]the public/private window oper-	
	ations in the MPI_WIN_SEPARATE memory model for two overlapping windows.	454
11.2	Active target communication	456
11.3	Active target communication, with weak synchronization	457
11.4	Passive target communication	458
		462
		482
11.7	Deadlock situation	482
		482
		512
		512
13.3	Displacements	525

13.4	Example array file layout	580
13.5	Example local array filetype for process 1	580
16.1	Status conversion routines	700

List of Tables

2.1	Deprecated [ticket0.341.]and removed constructs	19
$3.1 \\ 3.2 \\ 3.3$	Predefined MPI datatypes corresponding to Fortran datatypes Predefined MPI datatypes corresponding to C datatypes Predefined MPI datatypes corresponding to both C and Fortran datatypes	29 30 31
4.1	combiner values returned from MPI_TYPE_GET_ENVELOPE	125
6.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	270
$8.1 \\ 8.2$	Error classes (Part 1)	$\frac{366}{367}$
	C types of attribute value argument to MPI_Win_get_attr and MPI_Win_set_attr	$\begin{array}{c} 434\\ 471 \end{array}$
13.2	Data access routines	$527 \\ 562 \\ 578$
14.2 14.3 14.4	MPI tool information interface verbosity levels	591 592 594 598 618
16.2 16.3	C++ names for the MPI C and C++ predefined datatypes	634 634 635 680

- $46 \\ 47$

Acknowledgments

This document is the product of a number of distinct efforts in three distinct phases: one for each of MPI-1, MPI-2, and MPI-3. This section describes these in historical order, starting with MPI-1. Some efforts, particularly parts of MPI-2, had distinct groups of individuals associated with them, and these efforts are detailed separately.

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard.

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort.

Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are:

• Jack Dongarra, David Walker, Conveners and Meeting Chairs • Ewing Lusk, Bob Knighten, Minutes • Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication 24 • Al Geist, Marc Snir, Steve Otto, Collective Communication • Steve Otto, Editor • Rolf Hempel, Process Topologies • Ewing Lusk, Language Binding • William Gropp, Environmental Management • James Cownie, Profiling • Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators • Steven Huss-Lederman, Initial Implementation Subset The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

ticket0.

1	Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
2	Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
3	Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower
4	Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison
5	Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
6	Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
7	James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
8	Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
9	Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
10	Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
11	Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
12	Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith
13				
14	The University	v of Tennessee and ()ak Ridge National	Laboratory made the draft avail-
15	e e		0	tal in distributing the document.
16				t by ARPA and NSF under grant
17				d Technology Center Cooperative
18				ne European Community through
19	Esprit project P66	· · · · · · · · · · · · · · · · · · ·		ie European community inrough
20	Espire project i oo	10 (111L).		
21				
22		2.0.		
23	MPI-1.2 and MPI-	-2.0:		
24	Those who served	as primary coordina	tors in $MPI-1.2$ and	MPI-2.0 are:
25				
26	• Ewing Lusk,	Convener and Meet	ing Chair	
27 28	• Steve Huss-L	ederman, Editor		
29 30	• Ewing Lusk,	Miscellany		
31	• Bill Saphir, I	Process Creation and	d Management	
32 33	• Marc Snir, O	ne-Sided Communic	cations	
34	,			
35	• Bill Gropp a	nd Anthony Skjellur	n, Extended Collect	vive Operations
36	• Steve Huss-L	ederman, External	Interfaces	
37 38	• Bill Nitzberg	s, I/O		
39 40	• Andrew Lum	nsdaine, Bill Saphir,	and Jeff Squyres, L	anguage Bindings
40	A	-11		
42	• Anthony Skje	ellum and Arkady K	anevsky, Real-11me	2
42	The following	list includes some a	f the active particin	ants who attended MPI-2 Forum
43 44	0	not mentioned above		anto who attended WIF I-2 POLUIII
44	meetings and are i	to menuoned above		
45				
40				
48				
10				

Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar	1
Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz	2
Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen	3
Ying Chen	Albert Cheng	Yong Cho	Joel Clark	4
Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie	5
Zhenqian Cui	Suresh Damodaran-Kar	nal	Raja Daoud	6
Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra	7
Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon	8
Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman	9
Ian Foster	Hubertus Franke	Richard Frost	Al Geist	10
Robert George	David Greenberg	John Hagedorn	Kei Harada	11
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson	12
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones	13
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica	14
Steve Landherr	Mario Lauria	Mark Law	Juan Leon	15
Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams	16
John May	Oliver McBryan	Brian McCandless	Tyce McLarty	17
Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha	18
Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco	19
Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce	20
Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov	21
James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee	22
Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha	23
Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman	24
Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale	25
David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur	26
Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon	27
David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong	28
Dave Wright				29
The MDI Forum	also acknowledges and ap	provinted the valuable	a input from people via	30
e-mail and in person.	also acknowledges and ap	preciates the valuable	e input nom people via	31
e-man and m person.				32
The fellowing inc	titutions supported the	ADI 2 offerst through	time and there are ant	33
0	stitutions supported the \mathbb{N}	VIPI-2 enort through	time and travel support	34
for the people listed a	ibove.			35
Argonne Nationa	l Laboratory			36
Bolt, Beranek, an	nd Newman			37

Angonic National Laboratory	
Bolt, Beranek, and Newman	37
California Institute of Technology	38
Center for Computing Sciences	39
Convex Computer Corporation	40
Cray Research	41
Digital Equipment Corporation	42
Dolphin Interconnect Solutions, Inc.	43
Edinburgh Parallel Computing Centre	44
General Electric Company	45
German National Research Center for Information Technology	46
Hewlett-Packard	47
Hitachi	48

1	Hughes Aircraft Company
2	Intel Corporation
3	International Business Machines
4	Khoral Research
5	Lawrence Livermore National Laboratory
6	Los Alamos National Laboratory
7	MPI Software Techology, Inc.
8	Mississippi State University
9	NEC Corporation
10	National Aeronautics and Space Administration
11	National Energy Research Scientific Computing Center
12	National Institute of Standards and Technology
13	National Oceanic and Atmospheric Adminstration
14	Oak Ridge National Laboratory
15	Ohio State University
16	PALLAS GmbH
17	Pacific Northwest National Laboratory
18	Pratt & Whitney
19	San Diego Supercomputer Center
20	Sanders, A Lockheed-Martin Company
21	Sandia National Laboratories
22	Schlumberger
23	Scientific Computing Associates, Inc.
24	Silicon Graphics Incorporated
25	Sky Computers
26	Sun Microsystems Computer Corporation
27	Syracuse University
28	The MITRE Corporation
29	Thinking Machines Corporation
30	United States Navy
31 32	University of Colorado
33	University of Denver
33 34	University of Houston
35	University of Illinois
36	University of Maryland University of Natra Dama
37	University of Notre Dame
38	University of San Fransisco
39	University of Stuttgart Computing Center
40	University of Wisconsin
41	MDI 2 anomated on a name tight hudget (in prality it had no hudget when the first
42	MPI-2 operated on a very tight budget (in reality, it had no budget when the first meeting was appropriated). Many institutions helped the MPI 2 effort by supporting the
43	meeting was announced). Many institutions helped the MPI-2 effort by supporting the efforts and travel of the members of the MPI Forum. Direct support was given by NSF and
44	DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and
45	Esprit under project HPC Standards (21111) for European participants.
46	Laprie under project in C Standards (21111) for European participants.
47	

MPI-1.3 and MPI-2.1:	1
The editors and organizers of the combined documents have been:	2 3
• Richard Graham, Convener and Meeting Chair	4
• Jack Dongarra, Steering Committee	5 6
• Al Geist, Steering Committee	7
• Bill Gropp, Steering Committee	8 9
Rainer Keller, Merge of MPI-1.3	10
	11 12
• Andrew Lumsdaine, Steering Committee	13
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)	14 15 16
• Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)	17 18
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:	19 20 21
• Bill Gropp, Frontmatter, Introduction, and Bibliography	22
• Richard Graham, Point-to-Point Communication	23 24
• Adam Moody, Collective Communication	25 26
• Richard Treumann, Groups, Contexts, and Communicators	20
• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions	28 29 30
• George Bosilca, Environmental Management	31 32
• David Solt, Process Creation and Management	33
• Bronis R. de Supinski, External Interfaces, and Profiling	34 35
• Rajeev Thakur, I/O	36
	37 38
• Jeffrey M. Squyres, Language Bindings and MPI 2.1 Secretary	39
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log	40
• Alexander Supalov and Denis Nagorny, Annex Language Bindings	41 42
The following list includes some of the active participants who attended MPI-2 Forum	43 44
meetings and in the e-mail discussions of the errata items and are not mentioned above.	45
	46
	47 48
	-

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Jeffrey Brown
4	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
5	Terry Dontje	Gabor Dozsa	Edric Ellis
6	Karl Feind	Edgar Gabriel	Patrick Geoffray
7	David Gingold	Dave Goodell	Erez Haba
8	Robert Harrison	Thomas Herault	Steve Hodson
9	Torsten Hoefler	Joshua Hursey	Yann Kalemkarian
10	Matthew Koop	Quincey Koziol	Sameer Kumar
11	Miron Livny	Kannan Narasimhan	Mark Pagel
12	Avneesh Pant	Steve Poole	Howard Pritchard
13	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
14	Tony Skjellum	Brian Smith	Vinod Tipparaju
15	Jesper Larsson Träff	Keith Underwood	
16	The MPI Forum also acknowledges and appreciates the valuable input from people via		
17	e-mail and in person.		
18	c-man and m person.		
19	The following institutions supported the MPI-2 effort through time and travel support for the people listed above.		
20			
21	for the people listed at		
22	Argonne National Laboratory		
23	Bull		
24	Cisco Systems, Inc.		
25	Cray Inc.		
26	The HDF Group		
27	Hewlett-Packard		
28	IBM T.J. Watson Research		
29	Indiana University		
30	Institut National de Recherche en Informatique et Automatique (INRIA)		
31	Intel Corporation		
32	Lawrence Berkeley National Laboratory		
33	Lawrence Livermore National Laboratory		
34	Los Alamos National Laboratory		

- 35Mathworks 36
- Mellanox Technologies 37Microsoft
- 38Myricom
- 39NEC Laboratories Europe, NEC Europe Ltd.
- 40Oak Ridge National Laboratory
- 41Ohio State University
- Pacific Northwest National Laboratory 42
- QLogic Corporation 43
- Sandia National Laboratories 44
- 45SiCortex
- Silicon Graphics Incorporated 46
- 47Sun Microsystems, Inc.
- 48University of Alabama at Birmingham

University of Houston	1
University of Illinois at Urbana-Champaign University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	2 3
University of Tennessee, Knoxville	4
University of Wisconsin	5
	6 7
Funding for the MPI Forum meetings was partially supported by award $\#$ CCF-0816909	8
from the National Science Foundation. [In addition, the HDF Group provided travel support for one U.S. academic.	$_{9}$ ticket0.
port for one U.S. academic.	10 11
	12
MPI-2.2:	13
All chapters have been revisited to achieve a consistent MPI-2.2 text. Those who served as authors for the necessary modifications are:	14 15
• William Gropp, Frontmatter, Introduction, and Bibliography; MPI 2.2 chair.	16 17
	18
• Richard Graham, Point-to-Point Communication and Datatypes	19
• Adam Moody, Collective Communication	20 21
• Torsten Hoefler, Collective Communication and Process Topologies	22
• Richard Treumann, Groups, Contexts, and Communicators	23 24
• Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications	25
• George Bosilca, Datatypes and Environmental Management	26 27
• David Solt, Process Creation and Management	28 29
	30
• Bronis R. de Supinski, External Interfaces, and Profiling	31
• Rajeev Thakur, I/O	32 33
• Jeffrey M. Squyres, Language Bindings and MPI 2.2 Secretary	34
• Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language	35
Bindings	36 37
• Alexander Supalov, Annex Language Bindings	38
The following list includes some of the active participants who attended MPI-2 Forum	39
meetings and in the e-mail discussions of the errata items and are not mentioned above.	40 41
	42
	43
	44
	45 46
	47
	48

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Greg Bronevetsky
4	Jeff Brown	Darius Buntinas	Jonathan Carter
5	Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
6	Edric Ellis	Karl Feind	Edgar Gabriel
7	Patrick Geoffray	Johann George	David Gingold
8	David Goodell	Erez Haba	Robert Harrison
9	Thomas Herault	Marc-André Hermanns	Steve Hodson
10	Joshua Hursey	Yutaka Ishikawa	Bin Jia
11	Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
12	Ranier Keller	Matthew Koop	Quincey Koziol
13	Manojkumar Krishnan	Sameer Kumar	Miron Livny
14	Andrew Lumsdaine	Miao Luo	Ewing Lusk
15	Timothy I. Mattox	Kannan Narasimhan	Mark Pagel
16	Avneesh Pant	Steve Poole	Howard Pritchard
17	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
18	Martin Schulz	Pavel Shamis	Galen Shipman
19	Christian Siebert	Anthony Skjellum	Brian Smith
20	Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
21	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu
22	The MPI Forum also	acknowledges and appreciates	s the valuable input from people via
23	e-mail and in person.	acknowledges and approclate	s the valuable input from people via
24	e man and m person.		
25	The following institut	tions supported the MPI-2.2 ef	fort through time and travel support
26	for the people listed abov		
27	Argonne National La		
28	Auburn University	U U	
29	Bull		
30	Cisco Systems, Inc.		
31 32	Cray Inc.		
33	Forschungszentrum J	Jülich	
34	$\operatorname{Fujitsu}$		
35	The HDF Group		
36	Hewlett-Packard		
37	International Busines	ss Machines	
38	Indiana University		
39		Recherche en Informatique et	,
40		ed Science & Engineering Cor	poration
41	Intel Corporation		
42	Lawrence Berkeley National Laboratory		
43		National Laboratory	
44	Los Alamos National	l Laboratory	
45	Mathworks		
46	Mellanox Technologi	es	
47	Microsoft		
48	Myricom		

NEC Corporation	1
Oak Ridge National Laboratory	2
Ohio State University	3
Pacific Northwest National Laboratory	4
QLogic Corporation	5
RunTime Computing Solutions, LLC	6
Sandia National Laboratories	7
SiCortex, Inc.	8
Silicon Graphics Inc.	9
Sun Microsystems, Inc.	10
Tokyo Institute of Technology	11
University of Alabama at Birmingham	12
University of Houston	13
University of Illinois at Urbana-Champaign	14
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	15
University of Tennessee, Knoxville	16
University of Tokyo	17
University of Wisconsin	18
	19
Funding for the MPI Forum meetings was partially supported by award $\#$ CCF-0816909	
from the National Science Foundation. []In addition, the HDF Group provided travel sup-	$_{21}$ ticket0.
port for one U.S. academic.	$_{22}$ ticket0.
	23
	24
MPI-3:	25
	26
MPI-3 is a significant effort to extend and modernize the MPI Standard.	27
The editors and organizers of the MPI-3 have been: Taken from MPI-2.2 with minor	28
corrections. Need to separate the working groups list (which is currently reviewers) from the	29
primary authors . Also, did I miss active steering committee members?	30
• William Gropp, Steering committee, Frontmatter, Introduction, Groups, Contexts,	31
and Communicators, One-Sided Communications, and Bibliography	32
and communicators, one stated communications, and Bionography	33
• Richard Graham, Steering committee, Point-to-Point Communication; Meeting Con-	34
vener, and MPI-3 chair	

	30
Adam Moody, Collective Communication	
	37
• Torsten Hoefler, Collective Communication and Process Topologies	38
	39
• George Bosilca, Datatypes and Environmental Management	40
• David Solt, Process Creation and Management	41
• David Solt, 1 locess creation and Management	
• Bronis R. de Supinski, External Interfaces, and Profiling	43
	44
• Rajeev Thakur, I/O and One-Sided Communications	45

Darius Buntinas, Info ObjectJeffrey M. Squyres, Language Bindings and MPI 3.0 Secretary

• Rolf Rabenseifner, Steering committee, Terms and Definitions, Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings

• Craig Rasmussen, Fortran Bindings

The following list includes some of the active participants who attended MPI-3 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.

8	Sadaf Alam	Pavan Balaji	Purushotham V. Bangalore
9	Brian Barrett	Richard Barrett	Robert Blackmore
10	Ron Brightwell	Greg Bronevetsky	Darius Buntinas
11	James Dinan	Terry Dontje	Gabor Dozsa
12	Edgar Gabriel	David Goodell	Manjunath Gorentla Vankata
13	Erez Haba	Jeff Hammond	Thomas Herault
14	Marc-André Hermanns	Jennifer Herrett-Skjellum	Joshua Hursey
15	Yutaka Ishikawa	Bin Jia	Hideyuki Jitsumoto
16	Yann Kalemkarian	Chulho Kim	Christof Klausecker
17	Alice Koniges	Quincey Koziol	Dieter Kranzlmueller
18	Manojkumar Krishnan	Sameer Kumar	Andrew Lumsdaine
19	Miao Luo	Ewing Lusk	Kathryn Mohror
20	Steve Oyanagi	Mark Pagel	Steve Poole
21	Howard Pritchard	[ticket229.2.] Hubert Ritzdorf	
22	Timo Schneider	Martin Schulz	Christian Siebert
23	Anthony Skjellum	Brian Smith	Marc Snir
24	Shinji Sumimoto	Alexander Supalov	Sayantan Sur
25	Fabian Tillier	Vinod Tipparaju	Keith Underwood
26	Rolf Vandevaart	Abhinav Vishnu	

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

28 29 30

31

48

27

 $\mathbf{1}$

 $\mathbf{2}$

3

 $\frac{4}{5}$

6

7

ticket229.2.

The following institutions supported the MPI-3 effort through time and travel support for the people listed above.

32	Argonne National Laboratory
33	Bull
34	Cisco Systems, Inc.
35	Cray Inc.
36	CSCS
37	Forschungszentrum Jülich
38	Fujitsu
39	German Research School for Simulation Sciences
40	The HDF Group
41	Hewlett-Packard
42	
43	International Business Machines
44	IBM India Private Ltd
45	Indiana University
46	Institut National de Recherche en Informatique et Automatique (INRIA)
40	Institute for Advanced Science & Engineering Corporation
47	Intel Corporation

Unofficial Draft for Comment Only

]	Lawrence Berkeley National Laboratory	1
	Lawrence Livermore National Laboratory	2
]	Los Alamos National Laboratory	3
J	Ludwig-Maximilians Universität München	4
I	Microsoft	5
I	NEC Corporation	6
(Oak Ridge National Laboratory	7
-	The Ohio State University	8
(Oracle America	9
]	Pacific Northwest National Laboratory	10
(QLogic Corporation	11
J	RunTime Computing Solutions, LLC	12
Ş	Sandia National Laboratory	13
.	Technical University of Chemnitz	14
5	Tokyo Institute of Technology	15
I	University of Alabama at Birmingham	16
I	University of Chicago	17
I	University of Houston	18
I	University of Illinois at Urbana-Champaign	19
I	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	20
I	University of Tennessee, Knoxville	21
1	University of Tokyo	22
1	Funding for the MPI Forum meetings was partially supported by award $\#$ CCF-0816909	23
	the National Science Foundation. In addition, the HDF Group provided travel support	24
	ne U.S. academic.	25
101 01		26
		27
		28
		29
		30 31
		32
		33
		34
		35
		36
		37
		38
		39
		40
		41
		42
		43
		44
		45
		46
		47
		48

Chapter 1

Introduction to MPI

1.1 Overview and Goals

MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. [(]Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.[)] MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, [Fortran-77, and Fortran-95]and Fortran, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI's development.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a messagepassing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases [provide hardware support for]for which they can provide hardware support, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.

Unofficial Draft for Comment Only

 21

 24

ticket0.

ticket0.

ticket0.

ticket230-B.

- Allow convenient C, C++, [Fortran-77, and Fortran-95]and Fortran bindings for the interface.
 - Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
 - Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
 - Semantics of the interface should be language independent.
 - The interface should be designed to allow for thread safety.

1.2 Background of MPI-1.0

¹⁵ MPI sought to make use of the most attractive features of a number of existing message¹⁶ passing systems, rather than selecting one of them and adopting it as the standard. Thus,
¹⁷ MPI was strongly influenced by work at the IBM T. J. Watson Research Center [1, 2],
¹⁸ Intel's NX/2 [50], Express [13], nCUBE's Vertex [46], p4 [8, 9], and PARMACS [5, 10].
¹⁹ Other important contributions have come from Zipcode [53, 54], Chimp [17, 18], PVM
²⁰ [4, 15], Chameleon [27], and PICL [25].

21The MPI standardization effort involved about 60 people from 40 organizations mainly 22from the United States and Europe. Most of the major vendors of concurrent computers 23were involved in MPI, along with researchers from universities, government laboratories, and 24 industry. The standardization process began with the Workshop on Standards for Message-25Passing in a Distributed Memory Environment, sponsored by the Center for Research on 26Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [61]. At this workshop 27the basic features essential to a standard message-passing interface were discussed, and a 28working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [16]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

36 In November 1992, a meeting of the MPI working group was held in Minneapolis, at 37 which it was decided to place the standardization process on a more formal footing, and to 38generally adopt the procedures and organization of the High Performance Fortran Forum. 39 Subcommittees were formed for the major component areas of the standard, and an email 40discussion service established for each. In addition, the goal of producing a draft MPI 41 standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 426 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI 43standard at the Supercomputing 93 conference in November 1993. These meetings and the 44email discussion together constituted the MPI Forum, membership of which has been open 45to all members of the high performance computing community.

- 46
- 47
- 48

7 8

9

10

11 12 13

14

2

3

4

5 6

ticket230-B.¹

1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

2 Beginning in March 1995, the MPI Forum began meeting to consider corrections and exten-3 sions to the original MPI Standard document [22]. The first product of these deliberations 4 was Version 1.1 of the MPI specification, released in June of 1995 [23] (see 5http://www.mpi-forum.org for official MPI document releases). At that time, effort fo-6 cused in five areas. 7 8 1. Further corrections and clarifications for the MPI-1.1 document. 9 10 2. Additions to MPI-1.1 that do not significantly change its types of functionality (new 11 datatype constructors, language interoperability, etc.). 123. Completely new types of functionality (dynamic processes, one-sided communication, 13parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality." 14154. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 16and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 17 to handle Fortran 90 issues. 18 195. Discussions of areas in which the MPI process and framework seem likely to be useful, 20but where more discussion and experience are needed before standardization (e.g. 21zero-copy semantics on shared-memory machines, real-time specifications). 22Corrections and clarifications (items of type 1 in the above list) were collected in Chap-23ter 3 of the MPI-2 document: "Version 1.2 of MPI." That chapter also contains the function 24 for identifying the version number. Additions to MPI-1.1 (items of types 2, 3, and 4 in the 25above list) are in the remaining chapters of the MPI-2 document, and constitute the specifi-26cation for MPI-2. Items of type 5 in the above list have been moved to a separate document, 27the "MPI Journal of Development" (JOD), and are not part of the MPI-2 Standard. 28This structure makes it easy for users and implementors to understand what level of 29MPI compliance a given implementation has: 30 31 • MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of com-32 pliance. It means that the implementation conforms to the clarifications of MPI-1.1 33 function behavior given in Chapter 3 of the MPI-2 document. Some implementations 34 may require changes to be MPI-1 compliant. 3536 • MPI-2 compliance will mean compliance with all of MPI-2.1. 37 • The MPI Journal of Development is not part of the MPI Standard. 38 39

It is to be emphasized that forward compatibility is preserved. That is, a valid MPI-1.1 program is both a valid MPI-1.3 program and a valid MPI-2.1 program, and a valid MPI-1.3 program is a valid MPI-2.1 program.

1.4 Background of MPI-1.3 and MPI-2.1

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1" was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for 48

Unofficial Draft for Comment Only

1

40

41

42 43

MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done
 electronically. Both ballots were combined into one document: "Errata for MPI-2", May
 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors
 kept working on new requests for clarification.

5Restarting regular work of the MPI Forum was initiated in three meetings, at Eu-6 roPVM/MPI'06 in Bonn, at EuroPVM/MPI'07 in Paris, and at SC'07 in Reno. In De- $\overline{7}$ cember 2007, a steering committee started the organization of new MPI Forum meetings at 8 regular 8-weeks intervals. At the January 14-16, 2008 meeting in Chicago, the MPI Forum ticket0.⁹ decided to combine the existing and future MPI documents to one [single] document for each 10 version of the MPI standard. For technical and historical reasons, this series was started 11with MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started 12in 1995 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, 13Errata for MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft 14document, for each chapter, a chapter author and review team were defined. They cleaned 15up the document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard 16document was finished in June 2008, and finally released with a second vote in September 172008 in the meeting at Dublin, just before EuroPVM/MPI'08. The major work of the 18 current MPI Forum is the preparation of MPI-3. 19

1.5 Background of MPI-2.2

MPI-2.2 is a minor update to the MPI-2.1 standard. This version addresses additional errors and ambiguities that were not corrected in the MPI-2.1 standard as well as a small number of extensions to MPI-2.1 that met the following criteria:

- Any correct MPI-2.1 program is a correct MPI-2.2 program.
- Any extension must have significant benefit for users.
- Any extension must not require significant implementation effort. To that end, all such changes are accompanied by an open source implementation.

The discussions of MPI-2.2 proceeded concurrently with the MPI-3 discussions; in some cases, extensions were proposed for MPI-2.2 but were later moved to MPI-3.

1.6 Background of MPI-3.0

MPI-3.0 is a major update to the MPI standard. Areas of particular interest are the extension of collective operations to include nonblocking, with other areas under consideration. This *draft* contains the MPI Forum's current draft of nonblocking collective routines.

A new Fortran mpi_f08 module is introduced to provide extended compile-time argument checking and buffer handling in nonblocking routines. This new Fortran support method provides protection against the optimization problems[with] with asynchronous accesses to the buffers of nonblocking calls. The existing mpi module is enhanced to provide basic compile-time argument checking for MPI calls. The use of mpif.h is strongly discouraged.

48

20

21 22

23

 24

25 26

27

1.7 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran, C and C++. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

1.8 What Platforms Are Targets For Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the sharedand distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those "machines" consisting of collections of other machines, parallel or not, connected by a communication network.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogenous networks of workstations.

1.9 What Is Included In The Standard?

The standard includes:

- Point-to-point communication,
- Datatypes,
- Collective operations,
- Process groups,
- Communication contexts,
- Process topologies,
- Environmental [M]management and inquiry,
- The [i]Info object,
- Process creation and management,
- One-sided communication,

1 2

3

4

5

6

7

8 9 10

11 12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28 29

30 31

32 33

34

36

39

ticket0.

³⁵ ticket0.

 $_{37}$ ticket0.

 38 ticket0.

40 ticket0.

 $_{42}$ ticket0.

⁴³ ticket0.

 44 ticket0.

⁴⁵ ticket0.
 ⁴⁶ ticket0.

47 ticket0.

 48 ticket0.

ticket0.

1	• External interfaces,
ticket0. 3	• Parallel file I/O,
ticket0.	• Language [B] bindings for Fortran, C and C++,
ticket0. $^{\circ}_{7}$ ticket0. $^{6}_{7}$	• Profiling interface.
8 9	1.10 What Is Not Included In The Standard?
10 11	The standard does not specify:
12 13 14	• Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
15	• Program construction tools,
16 17	• Debugging facilities.
18 19 20 21 22 23	There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self-imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.
24 25 26	1.11 Organization of this Document
26 27 28	The following is a list of the remaining chapters in this document, along with a brief description of each.
29 30 31	• Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
32 33 34 35	• Chapter 3, Point to Point Communication, defines the basic, pairwise communication subset of MPI. <i>Send</i> and <i>receive</i> are found here, along with many associated functions designed to make basic communication powerful and efficient.
36 37	• Chapter 4, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.
$^{38}_{39}$ 40 41 ticket0. $^{42}_{43}$	• Chapter 5, Collective Communications, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations.
44 45 46 47	• Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a <i>communicator</i> .
48	

 $\mathbf{6}$

- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 9, The Info Object, defines an opaque object, that is used as input [of]in several MPI routines.
- Chapter 10, Process Creation and Management, defines routines that allow for creation of processes.
- Chapter 11, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 12, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 13, I/O, defines MPI support for parallel I/O.
- Chapter 14.2, Profiling Interface, explains a simple name-shifting convention that any MPI implementation must support. One motivation for this is the ability to put performance profiling calls into MPI without the need for access to the MPI source code. The name shift is merely an interface, it says nothing about how the actual profiling should be done and in fact, the name shift can be useful for other purposes.
- Chapter 15, Deprecated Functions, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 16, Language Bindings, describes the C++ binding, discusses Fortran issues, and describes language interoperability aspects between C, C++, and Fortran.

The Appendices are:

- Annex A, Language Bindings Summary, gives specific syntax in C, C++, and Fortran, for all MPI functions, constants, and types.
- Annex B, Change-Log, summarizes major changes since the previous version of the standard.
- Several Index pages [are showing]show the locations of examples, constants and predefined handles, callback routine[s'] prototypes, and all MPI functions.

 MPI provides various interfaces to facilitate interoperability of distinct MPI implementations. Among these are the canonical data representation for MPI I/O and for

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

 $45 \\ 46$

tick	et0.
tick	et0.

 $_{10}$ ticket0.

MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual bind- $\mathbf{2}$ ing of these interfaces that will enable interoperability is outside the scope of this document. A separate document consists of ideas that were discussed in the MPI Forum and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are • Chapter 2, Spawning Independent Processes, includes some elements of dynamic pro-cess management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard. • Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment. • Chapter 4, Communicator ID, describes an approach to providing identifiers for com-municators. • Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particu-lar single-copy routines for use in shared-memory environments and new datatype constructors. • Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface. 24 • Chapter 7, Split Collective Communication, describes a specification for certain non-blocking collective operations. • Chapter 8, Real-Time MPI, discusses MPI support for real time processing. 31

Chapter 2

MPI Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices. It is similar to the MPI-1 Terms and Conventions chapter but differs in some major and minor ways. Some of the major areas of difference are the naming conventions, some semantic definitions, file objects, Fortran 90 vs Fortran 77, C++, processes, and interaction with signals.

2.1 Document Notation

Rationale. Throughout this document, the rationale for the design choices made in the interface specification is set off in this format. Some readers may wish to skip these sections, while readers interested in interface design may want to read them carefully. (*End of rationale.*)

Advice to users. Throughout this document, material aimed at users and that illustrates usage is set off in this format. Some readers may wish to skip these sections, while readers interested in programming in MPI may want to read them carefully. (End of advice to users.)

Advice to implementors. Throughout this document, material that is primarily commentary to implementors is set off in this format. Some readers may wish to skip these sections, while readers interested in MPI implementations may want to read them carefully. (End of advice to implementors.)

2.2 Naming Conventions

In many cases MPI names for C functions are of the form MPI_Class_action_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules. The C++ bindings in particular follow these rules (see Section 2.6.4 on page 20).

1. In C, all routines associated with a particular type of MPI object should be of the form MPI_Class_action_subset or, if no subset exists, of the form MPI_Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI_CLASS_ACTION_SUBSET or, if no subset exists, of the form

 24

	10 CHAPTER 2. MPI TERMIS AND CONVENTIONS
1 2 3 4 5	MPI_CLASS_ACTION. For C and Fortran we use the C++ terminology to define the Class. In C++, the routine is a method on Class and is named MPI::Class::Action_subset. If the routine is associated with a certain class, but does not make sense as an object method, it is a static member function of the class.
6 7 8	 If the routine is not associated with a class, the name should be of the form MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran, and in C++ should be scoped in the MPI namespace, MPI::Action_subset.
9 10 11 12	3. The names of certain actions have been standardized. In particular, Create creates a new object, Get retrieves information about an object, Set sets this information, Delete deletes information, Is asks whether or not an object has a certain property.
13 14 15 16	C and Fortran names for some MPI functions (that were defined during the MPI-1 process) violate these rules in several cases. The most common exceptions are the omission of the Class name from the routine and the omission of the Action where one can be inferred.
17 18 19	MPI identifiers are limited to 30 characters (31 with the profiling interface). This is done to avoid exceeding the limit on some compilation systems.
20 21	2.3 Procedure Specification
22 23 24	MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT or INOUT. The meanings of these are:
$ticket 140. \frac{25}{26}$	• IN: the call may use the input value but does not update the argument from the perspective of the caller at any time during the call's execution,
28	• OUT: the call may update the argument but does not use its input value,
29 30	• INOUT: the call may both use and update the argument.
31 32 33 34 35 36 37	There is one special case — if an argument is a handle to an opaque object (these terms are defined in Section 2.5.1), and the object is updated by the procedure call, then the argument is marked INOUT or OUT. It is marked this way even though the handle itself is not modified — we use the INOUT or OUT attribute to denote that what the handle <i>references</i> is updated. Thus, in C++, IN arguments are usually either references or pointers to const objects.
38 39 40 41	<i>Rationale.</i> The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (<i>End of rationale.</i>)
42 43 44 45 46 47 48	MPI's use of IN, OUT and INOUT is intended to indicate to the user how an argument is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., INTENT in Fortran 90 bindings or const in C bindings). For instance, the "constant" MPI_BOTTOM can usually be passed to OUT buffer arguments. Similarly, MPI_STATUS_IGNORE can be passed as the OUT status argument. A common occurrence for MPI functions is an argument that is used as IN by some pro- cesses and OUT by other processes. Such an argument is, syntactically, an INOUT argument

CHAPTER 2. MPI TERMS AND CONVENTIONS

10

and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process then an arbitrary value can be passed as an argument.

Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased with any other argument passed to an MPI procedure. An example of argument aliasing in C appears below. If we define a C procedure like this,

```
void copyIntBuffer( int *pin, int *pout, int len )
{    int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}</pre>
```

then a call to it in the following code fragment has aliased arguments.

```
int a[10];
copyIntBuffer( a, a+3, 7);
```

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified. Note that Fortran prohibits aliasing of arguments.

All MPI functions are first specified in the language-independent notation. Immediately below this, [the ISO C version of the function is shown followed by a version of the same function in Fortran and then the C++ binding.]language dependent bindings follow:

- The ISO C version of the function.
- The Fortran version used with USE mpi_f08.
- The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'
- The C++ binding (which is deprecated).

"Fortran" in this document refers to Fortran 90 and higher; see Section 2.6.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

nonblocking A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is **started** by the call that initiates it, e.g., MPI_ISEND. The word complete is used with respect to operations, requests, and communications. An **operation completes** when the user is allowed to reuse resources, and any output buffers have been updated; i.e. a call to MPI_TEST will return flag = true. A **request is completed** by a call to wait, which returns, or a test or get status call which returns flag = true. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is **freed**, and becomes **inactive** if it was persistent. A **communication completes** when all participating operations complete.

Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8 9

10

11

12

13 14

15 16

17 18

19

20

21

23 24

25 26

27

28 29

30

32 33

34 35

36 37

38

39

40

41

42

43

44

45

46

47

48

₂₂ ticket230-B.

³¹ ticket230-B.

1 2	blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.
3 4 5	local A procedure is local if completion of the procedure depends only on the local executing process.
6 7 8 9	non-local A procedure is non-local if completion of the operation may require the exe- cution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.
10 11 12 13	collective A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronizing. Collective calls over the same communicator must be executed in the same order by all members of the process group.
14 15 16 17 18 19	<pre>predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT_INT, or MPI_UB) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are named whereas the latter are unnamed.</pre>
20 21	derived A derived datatype is any datatype that is not predefined.
22 23 24 25 26 27 28 29 30 ticket280. ³¹ 32 33 34 35	 portable A datatype is portable, if it is a predefined datatype, or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different
36 37	architecture.
38 39 40 41	equivalent Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

42Data Types 2.5

43

Opaque Objects 442.5.1

45MPI manages system memory that is used for buffering messages and for storing internal 46 representations of various MPI objects such as groups, communicators, datatypes, etc. This 47memory is not directly accessible to the user, and objects stored there are **opaque**: their 48

size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

In Fortran with USE mpi or INCLUDE 'mpif.h', all handles have type INTEGER. In Fortran with USE mpi_f08, and in C and C++, a different handle type is defined for each category of objects. With Fortran USE mpi_f08, the handles are defined as Fortran BIND(C) derived types that consist of only one element INTEGER :: MPI_VAL. The internal handle value is identical to the Fortran INTEGER value used in the mpi module and mpif.h. The type names are identical to the names in C, except that they are not case sensitive. For example:

```
TYPE, BIND(C) :: MPI_Comm
INTEGER :: MPI_VAL
END TYPE MPI_Comm
```

In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. C++ handles can simply "wrap up" a table index or pointer. (*End of advice to implementors.*)

Rationale. Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle COMM can be converted directly into an exactly equivalent mpi_f08 communicator handle named comm_f08 by comm_f08%MPI_VAL=COMM, and vice versa. The use of the INTEGER defined handles and the BIND(C) derived type handles is different: Fortran 2003 (and later) define that BIND(C) derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these BIND(C) derived type handles. Most compilers use one unit for both, the INTEGER handles and the handles defined as BIND(C) derived types. (End of rationale.)

Advice to users. If a user wants to substitute mpif.h or the mpi module by the mpi_f08 module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (*End of advice to users.*)

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the objects are described. The calls accept a handle argument of matching type. In an allocate call this is an OUT argument that returns a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an "invalid handle" value. MPI provides an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not

Unofficial Draft for Comment Only

deallocate the object immediately. Any operation pending (at the time of the deallocate)
 that involves this object will complete normally; the object will be deallocated afterwards.

³ An opaque object and its handle are significant only at the process where the object ⁴ was created and cannot be transferred to another process.

⁵ MPI provides certain predefined opaque objects and predefined, static handles to these ⁶ objects. The user must not free such objects. In C++, this is enforced by declaring the ⁷ handles to these predefined objects to be static const.

Rationale. This design hides the internal representation used for MPI data structures, thus allowing similar calls in C, C++, and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows future extensions of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

- The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. (*End of rationale.*)
- Advice to users. A user may accidentally create a dangling reference by assigning to a 27handle the value of another handle, and then deallocating the object associated with 28 these handles. Conversely, if a handle variable is deallocated before the associated 29 object is freed, then the object becomes inaccessible (this may occur, for example, if 30 the handle is a local variable within a subroutine, and the subroutine is exited before 31 the associated object is deallocated). It is the user's responsibility to avoid adding or 32 deleting references to opaque objects, except as a result of MPI calls that allocate or 33 deallocate such objects. (End of advice to users.) 34
- 35 Advice to implementors. The intended semantics of opaque objects is that opaque 36 objects are separate from one another; each call to allocate such an object copies 37 all the information required for the object. Implementations may avoid excessive 38 copying by substituting referencing for copying. For example, a derived datatype 39 may contain references to its components, rather then copies of its components; a 40 call to MPI_COMM_GROUP may return a reference to the group associated with the 41 communicator, rather than a copy of this group. In such cases, the implementation 42must maintain reference counts, and allocate and deallocate objects in such a way that 43 the visible effect is as if the objects were copied. (End of advice to implementors.) 44

2.5.2 Array Arguments

45

46

⁴⁷ An MPI call may need an argument that is an array of opaque objects, or an array of ⁴⁸ handles. The array-of-handles is a regular array with entries that are handles to objects

Unofficial Draft for Comment Only

8

9

10

11

12

of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array. The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one uses MPI_STATUSES_IGNORE.

2.5.3 State

MPI procedures use at various places arguments with *state* types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values MPI_ORDER_C and MPI_ORDER_FORTRAN.

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions (Chapter 7 of the MPI-1 document). The range of other values, such as source, depends on values given by other MPI routines (in the case of source it is the communicator size).

MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments, but not necessarily in array declarations or as labels in C/C++ switch or Fortran select/case statements. This implies named constants to be link-time but not necessarily compile-time constants. The named constants listed below are required to be compile-time constants in both C/C++ and Fortran. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE). The handles themselves are constants and can be also used in initialization expressions or assignments.

The constants that are required to be compile-time constants (and can thus be used for array length declarations and labels in C/C++ switch and Fortran case/select statements) are:

MPI_MAX_PROCESSOR_NAME MPI_MAX_LIBRARY_VERSION_STRING MPI_MAX_ERROR_STRING MPI_MAX_DATAREP_STRING MPI_MAX_INFO_KEY MPI_MAX_INFO_VAL MPI_MAX_OBJECT_NAME MPI_MAX_PORT_NAME MPI_STATUS_SIZE (Fortran only) MPI_ADDRESS_KIND (Fortran only) $^{38}_{39}$ ticket204.



1

2

3

4

5 6

7

8 9

10

11

12

13

14 15

16 17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

	1	MPI_COUNT_KIND (Fortran only)
	2	MPI_INTEGER_KIND (Fortran only)
ticket234-F.		MPI_OFFSET_KIND (Fortran only)
ticket238-J.		MPI_SUBARRAYS_SUPPORTED (Fortran only)
ticket229.1.		MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
	6 7	and their C++ counterparts where appropriate.
	8	The constants that cannot be used in initialization expressions or assignments in For-
	9	tran are:
	10	MPI_BOTTOM
	11	MPI_STATUS_IGNORE
	12	MPI_STATUSES_IGNORE
	13	MPI_ERRCODES_IGNORE
	14	MPI_IN_PLACE
	15	MPI_ARGV_NULL MPI_ARGVS_NULL
	16	MPI_UNWEIGHTED
	17	MF1_ONWEIGHTED
:	18	Advice to implementors. In Fortran the implementation of these special constants
:	19	may require the use of language constructs that are outside the Fortran standard.
:	20	Using special values for the constants (e.g., by defining them through PARAMETER
	21	statements) is not possible because an implementation cannot distinguish these val-
ticket182.	22	ues from [legal]valid data. Typically, these constants are implemented as predefined
1	23	static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact
1	24	that the target compiler passes data by address. Inside the subroutine, this address
:	25	can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran
1	26	extensions or by implementing the function in C). (End of advice to implementors.)
:	27	
2	28	2.5.5 Choice
	29	MPI functions sometimes use arguments with a <i>choice</i> (or union) data type. Distinct calls to
:	30	the same routine may pass by reference actual arguments of different types. The mechanism
1:1 1094 E	31	for providing such arguments will differ from language to language. For Fortran with the
	32	include file mpif.h or the mpi module, the document uses <type> to represent a choice</type>
ticket234-F.	33	variable; with the Fortran mpi_f08 module, such arguments are declared with the Fortran
ticket234-F.		2008 + TR 29113 syntax TYPE(*), DIMENSION(); for C and C++, we use void *.
	36	Advice to implementors. Implementors can freely choose how to implement choice
	37	arguments in the mpi module, e.g., with a non-standard compiler-dependent method
	38 20	that has the quality of the call mechanism in the implicit Fortran interfaces, or with
	39 40	the method defined for the mpi_f08 module. See details in Section 16.2.1 on page 642.
	40	(End of advice to implementors.)
	42	
	43	
	44	2.5.6 Addresses
	45	Some MPI procedures use <i>address</i> arguments that represent an absolute address in the
	46	calling program. The datatype of such an argument is MPI_Aint in C, MPI::Aint in C++
	47	and INTEGER (KIND=MPI_ADDRESS_KIND) in Fortran. These types must have the same

and INTEGER (KIND=MPI_ADDRESS_KIND) in Fortran. These types must have the same
 width and encode address values in the same manner such that address values in one

language may be passed directly to another language without conversion. There is the MPI constant MPI_BOTTOM to indicate the start of the address range.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be INTEGER (KIND=MPI_OFFSET_KIND) in Fortran. In C one uses MPI_Offset whereas in C++ one uses MPI::Offset. These types must have the same width and encode address values in the same manner such that offset values in one language may be passed directly to another language without conversion.

2.5.8 Counts

As described above, MPI defines types (e.g., MPI_Aint) to address locations within memory and other types (e.g., MPI_Offset) to address locations within files. In addition, some MPI procedures use *count* arguments that represent a number of MPI datatypes on which to operate. At times, one needs a single type that can be used to address locations within either memory or files as well as express *count* values, and that type is MPI_Count in C and INTEGER (KIND=MPI_COUNT_KIND) in Fortran. These types must have the same width and encode values in the same manner such that count values in one language may be passed directly to another language without conversion. The size of the MPI_Count type is determined by the MPI implementation with the restriction that it must be minimally capable of encoding any value that may be stored in a variable of type int, MPI_Aint, or MPI_Offset in C and of type INTEGER, INTEGER (KIND=MPI_ADDRESS_KIND), or INTEGER (KIND=MPI_OFFSET_KIND) in Fortran.

Rationale. Count values logically need to be large enough to encode any value used for expressing element counts, type maps in memory, type maps in file views, etc. For backward compatibility reasons, many MPI routines still use **int** in C and **INTEGER** in Fortran as the type of count arguments. (*End of rationale.*)

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) The C++ language bindings have been deprecated. Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90 or later, though they [are]were originally designed to be usable in Fortran 77 environments. With the mpi_f08 module, two new Fortran features, *assumed type* and *assumed rank*, are also required, see Section 2.5.5 on page 16.

Since the word PARAMETER is a keyword in the Fortran language, we use the word "argument" to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will understand the word "argument" (which has no specific meaning in C/C++), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Unofficial Draft for Comment Only

 $\overline{7}$

 24

11 ticket265.

⁴⁰ ticket234-F.



Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid the "mpi_" and "pmpi_" prefixes.

ticket0.341.

⁵ 2.6.1 Deprecated and Removed Names and Functions

ticket0.341. 7 A number of chapters refer to deprecated or replaced [MPI-1]MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15 8 ticket0.341. 9 on page 619, but that users are recommended not to continue using, since better solutions ticket0.341. 10 were provided with [MPI-2] newer versions of MPI. For example, the Fortran binding for MPI-1 functions that have address arguments uses INTEGER. This is not consistent with the 11 C binding, and causes problems on machines with 32 bit INTEGERs and 64 bit addresses. 12In MPI-2, these functions were given new names with new bindings for the address argu-13 ments. The use of the old functions is deprecated. For consistency, here and in a few other 14cases, new C functions are also provided, even though the new functions are equivalent 15to the old functions. The old names are deprecated. Another example is provided by the 16MPI-1 predefined datatypes MPI_UB and MPI_LB. They are deprecated, since their use is 17awkward and error-prone. The MPI-2 function MPI_TYPE_CREATE_RESIZED provides a 18 more convenient mechanism to achieve the same effect. 19

ticket0.341. 20 ticket0.341. 21 22 Table 2.1 shows a list of all of the deprecated and removed constructs. Some of the deprecated constructs are now removed, but still listed here and in Chapter ?? on page ??. Note that the constants MPI_LB and MPI_UB are replaced by the function

> MPI_TYPE_CREATE_RESIZED; this is because their principal use was as input datatypes to MPI_TYPE_STRUCT to create resized datatypes. Also note that some C typedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term "Fortran" is used it means Fortran 90 or later; it means Fortran 2008 + TR 29113 and later if the mpi_f08 module is used.

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare variables, parameters, or functions with names beginning with the prefix MPI_. To avoid conflicting with the profiling interface, programs should also avoid functions with the prefix PMPI_. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. With USE mpi_f08, this last argument is declared as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g.,

MPI_NULL_COPY_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C and C++ as discussed in Section 16.3.9.

Handles are represented in Fortran as INTEGERs, or as a BIND(C) derived type with the mpi_f08 module; see Section 2.5.1 on page 12. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

1

2

3

4

23

 24

25 26

27 28

29

30

31

32

33

34

35

36

37

38

40

41

42

43

44

45

46

47

48

ticket234-F

ticket239-K.

ticket250-V.

ticket239-K.

ticket231-C.

[ticket0.341.]	et0.341.]
----------------	-----------

Deprecated or removed	deprecated	removed	Replacement	3
construct	since	since	L	4
MPI_ADDRESS	MPI-2.0		MPI_GET_ADDRESS	5
MPI_TYPE_HINDEXED	MPI-2.0		MPI_TYPE_CREATE_HINDEXED	6
MPI_TYPE_HVECTOR	MPI-2.0		MPI_TYPE_CREATE_HVECTOR	7
MPI_TYPE_STRUCT	MPI-2.0		MPI_TYPE_CREATE_STRUCT	°
MPI_TYPE_EXTENT	MPI-2.0		MPI_TYPE_GET_EXTENT	9
MPI_TYPE_UB	MPI-2.0		MPI_TYPE_GET_EXTENT	
MPI_TYPE_LB	MPI-2.0		MPI_TYPE_GET_EXTENT	10
MPI_LB ¹	MPI-2.0		MPI_TYPE_CREATE_RESIZED	11
MPI_UB ¹	MPI-2.0		MPI_TYPE_CREATE_RESIZED	12
MPI_ERRHANDLER_CREATE	MPI-2.0		MPI_COMM_CREATE_ERRHANDLER	13
MPI_ERRHANDLER_GET	MPI-2.0		MPI_COMM_GET_ERRHANDLER	14
MPI_ERRHANDLER_SET	MPI-2.0		MPI_COMM_SET_ERRHANDLER	15
$MPI_Handler_function^2$	MPI-2.0		MPI_Comm_errhandler_function ²	16
MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL	
MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL	17
MPI_DUP_FN ³	MPI-2.0		MPI_COMM_DUP_FN ³	18
MPI_NULL_COPY_FN ³	MPI-2.0		MPI_COMM_NULL_COPY_FN ³	19
MPI_NULL_DELETE_FN ³	MPI-2.0		MPI_COMM_NULL_DELETE_FN ³	20
MPI_Copy_function ²	MPI-2.0		MPI_Comm_copy_attr_function ²	21
COPY_FUNCTION ³	MPI-2.0		COMM_COPY_ATTR_[ticket250-V.][FN]FU	NGTION ³
MPI_Delete_function ²	MPI-2.0		MPI_Comm_delete_attr_function ²	23
DELETE_FUNCTION ³	MPI-2.0		COMM_DELETE_ATTR_[ticket250-V.][FN]I	
MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR	24
MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR	25
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR	26
MPI::	MPI-2.2	MPI-3.0	C language binding	27
¹ Predefined datatype.				28
² Callback prototype definition				29
³ Predefined callback routine.				30
Other entries are regular MPI	routines.			31
~				32

Table 2.1: Deprecated [ticket0.341.] and removed constructs

The older MPI Fortran binding is s (mpif.h and use mpi) are inconsistent with the Fortran 90 standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 16.2.16. [They are also inconsistent with Fortran 77.]

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an MPI_ prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare variables or functions with names beginning with the prefix MPI_. To support the profiling interface, programs should not declare functions with names beginning with the prefix PMPI_.

The definition of named constants, function prototypes, and type definitions must be

Unofficial Draft for Comment Only

1

3435

36

37

38

40

4142

43

44

45

46

47

48

ticket229.2.

ticket230-B.

ticket229.2.

ticket230-B.

³⁹ ticket230-B.

1 supplied in an include file mpi.h. $\mathbf{2}$ Almost all C functions return an error code. The successful return code will be 3 MPI_SUCCESS, but failure return codes are implementation dependent. 4 Type declarations are provided for handles to each category of opaque objects. 5Array arguments are indexed from zero. 6 Logical flags are integers with value 0 meaning "false" and a non-zero value meaning $\overline{7}$ "true." 8 Choice arguments are pointers of type void *. 9 Address arguments are of MPI defined type MPI_Aint. File displacements are of type 10 MPI_Offset. MPI_Aint is defined to be an integer of the size needed to hold any valid address 11on the target architecture. MPI_Offset is defined to be an integer of the size needed to hold 12any valid file size on the target architecture. 13 14C++ Binding Issues 2.6.4 15The C++ language bindings have been deprecated. There are places in the standard that 16give rules for C and not for C++. In these cases, the C rule should be applied to the C++17case, as appropriate. In particular, the values of constants given in the text are the ones 18 for C and Fortran. A cross index of these with the C++ names is given in Annex A. 19 We use the ISO C++ declaration format. All MPI names are declared within the scope 20of a namespace called MPI and therefore are referenced with an MPI:: prefix. Defined 21constants are in all capital letters, and class names, defined types, and functions have only 22 their first letter capitalized. Programs must not declare variables or functions in the MPI 23namespace. This is mandated to avoid possible name collisions. 24 The definition of named constants, function prototypes, and type definitions must be 25supplied in an include file mpi.h. 2627The file mpi.h may contain both the C and C++ defini-Advice to implementors. 28tions. Usually one can simply use the defined value (generally __cplusplus, but not 29 required) to see if one is using C++ to protect the C++ definitions. It is possible 30 ticket182. that a C compiler will require that the source protected this way be [legal]valid C 31 code. In this case, all the C++ definitions can be placed in a different include file and 32 the "#include" directive can be used to include the necessary C++ definitions in the 33 mpi.h file. (End of advice to implementors.) 3435 C++ functions that create objects or return information usually place the object or 36 information in the return value. Since the language neutral prototypes of MPI functions 37 include the C++ return value as an OUT parameter, semantic descriptions of MPI functions 38refer to the C++ return value by that parameter name. The remaining C++ functions 39 return void. 40In some circumstances, MPI permits users to indicate that they do not want a return 41 value. For example, the user may indicate that the status is not filled in. Unlike C and 42Fortran where this is achieved through a special input value, in C++ this is done by having 43two bindings where one has the optional argument and one does not. 44C++ functions do not return error codes. If the default error handler has been set 45to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an 46 error by throwing an MPI::Exception object.

⁴⁷ It should be noted that the default error handler (i.e., MPI::ERRORS_ARE_FATAL) on a ⁴⁸ given type has not changed. User error handlers are also permitted. MPI::ERRORS_RETURN

1

2

3

4

5 6

 $\overline{7}$

8

9

10 11

12

13

14

15

16

17

18

19

20

21

22

33

34

35 36

37 38

39

40

41

42 43

44

45

46

47

48

simply returns control to the calling function; there is no provision for the user to retrieve the error code.

User callback functions that return integer error codes should not throw exceptions; the returned error will be handled by the MPI implementation by invoking the appropriate error handler.

Advice to users. C++ programmers that want to handle MPI errors on their own should use the MPI::ERRORS_THROW_EXCEPTIONS error handler, rather than MPI::ERRORS_RETURN, that is used for that purpose in C. Care should be taken using exceptions in mixed language situations. (*End of advice to users.*)

Opaque object handles must be objects in themselves, and have the assignment and equality operators overridden to perform semantically like their C and Fortran counterparts.

Array arguments are indexed from zero.

Logical flags are of type bool.

Choice arguments are pointers of type void *.

Address arguments are of MPI-defined integer type MPI::Aint, defined to be an integer of the size needed to hold any valid address on the target architecture. Analogously, MPI::Offset is an integer to hold file offsets.

Most MPI functions are methods of MPI C++ classes. MPI class names are generated from the language neutral MPI types by dropping the MPI_ prefix and scoping the type within the MPI namespace. For example, MPI_DATATYPE becomes MPI::Datatype.

The names of MPI functions generally follow the naming rules given. In some circum-23stances, the MPI function is related to a function defined already for MPI-1 with a name 24 that does not follow the naming conventions. In this circumstance, the language neutral 25name is in analogy to the MPI name even though this gives an MPI-2 name that violates the 26naming conventions. The C and Fortran names are the same as the language neutral name 27in this case. However, the C++ names do reflect the naming rules and can differ from the C 28and Fortran names. Thus, the analogous name in C++ to the MPI name may be different 29than the language neutral name. This results in the C++ name differing from the language 30 neutral name. An example of this is the language neutral name of MPI_FINALIZED and a 31 C++ name of MPI::ls_finalized. 32

In C++, function typedefs are made publicly within appropriate classes. However, these declarations then become somewhat cumbersome, as with the following:

{typedef MPI:::Grequest::Query_function(); (binding deprecated, see Section 15.2)}

would look like the following:

```
namespace MPI {
   class Request {
        // ...
   };
   class Grequest : public MPI::Request {
        // ...
      typedef Query_function(void* extra_state, MPI::Status& status);
   };
};
```

1 2 3	Rather than including this scaffolding when declaring $C++$ typedefs, we use an abbreviated form. In particular, we explicitly indicate the class and namespace scope for the typedef of the function. Thus, the example above is shown in the text as follows:
4 5 6	<pre>typedef int MPI::Grequest::Query_function(void* extra_state,</pre>
7 8 9 10 11	The C++ bindings presented in Annex A.5 and throughout this document were gener- ated by applying a simple set of name generation rules to the MPI function specifications. While these guidelines may be sufficient in most cases, they may not be suitable for all situations. In cases of ambiguity or where a specific semantic statement is desired, these guidelines may be superseded as the situation dictates.
12 13 14	1. All functions, types, and constants are declared within the scope of a namespace called MPI.
15 16	2. Arrays of MPI handles are always left in the argument list (whether they are IN or OUT arguments).
17 18 19 20 21 22	3. If the argument list of an MPI function contains a scalar IN handle, and it makes sense to define the function as a method of the object corresponding to that handle, the function is made a member function of the corresponding MPI class. The member functions are named according to the corresponding MPI function name, but without the "MPI_" prefix and without the object name prefix (if applicable). In addition:
23 24 25 26	(a) The scalar IN handle is dropped from the argument list, and this corresponds to the dropped argument.(b) The function is declared const.
27 28 29	4. MPI functions are made into class functions (static) when they belong on a class but do not have a unique scalar IN or INOUT parameter of that class.
30 31 32 33	5. If the argument list contains a single OUT argument that is not of type MPI_STATUS (or an array), that argument is dropped from the list and the function returns that value.
34 35	Example 2.1 The C++ binding for MPI_COMM_SIZE is int MPI::Comm::Get_size(void) const.
36 37 38	6. If there are multiple OUT arguments in the argument list, one is chosen as the return value and is removed from the list.
39 40	7. If the argument list does not contain any OUT arguments, the function returns void.
41 42 43	Example 2.2 The C++ binding for MPI_REQUEST_FREE is void MPI::Request::Free(void)
44 45 46	8. MPI functions to which the above rules do not apply are not members of any class, but are defined in the MPI namespace.
47 48	Example 2.3 The C++ binding for MPI_BUFFER_ATTACH is void MPI::Attach_buffer(void* buffer, int size).

1

2

3

4 5

6

7

8 9 10

11

12

13

1415

16

17 18

19

2021

2223

 24

25

26

27

28

29

30

 31

32

33

35

36

37 38

39

40

41 42

43 44

4546

47

48

- 9. All class names, defined types, and function names have only their first letter capitalized. Defined constants are in all capital letters.
- 10. Any IN pointer, reference, or array argument must be declared const.
- 11. Handles are passed by reference.
- 12. Array arguments are denoted with square brackets ([]), not pointers, as this is more semantically precise.

2.6.5 Functions and Macros

An implementation is allowed to implement MPI_WTIME, MPI_WTICK, PMPI_WTIME, PMPI_WTICK, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 16.3.4, and no others, as macros in C.

Advice to implementors. Implementors should document which routines are implemented as macros. (End of advice to implementors.)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (End of advice to users.)

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process 34 signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (End of advice to implementors.)

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other

error conditions. In other words, MPI does not provide mechanisms for dealing with failures
 in the communication system. If the MPI implementation is built on an unreliable underly ing mechanism, then it is the job of the implementor of the MPI subsystem to insulate the
 user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible,
 such failures will be reflected as errors in the relevant communication call. Similarly, MPI
 itself provides no mechanisms for handling processor failures.

 $\overline{7}$ Of course, MPI programs may still be erroneous. A **program error** can occur when 8 an MPI call is made with an incorrect argument (non-existing destination in a send oper-9 ation, buffer too small in a receive operation, etc.). This type of error would occur in any 10 implementation. In addition, a **resource error** may occur when a program exceeds the 11amount of available system resources (number of pending messages, system buffers, etc.). 12The occurrence of this type of error depends on the amount of available resources in the 13system and the resource allocation mechanism used: this may differ from system to system. 14A high-quality implementation will provide generous limits on the important resources so 15as to alleviate the portability problem this represents.

16In C and Fortran, almost all MPI calls return a code that indicates successful completion 17of the operation. Whenever possible, MPI calls return an error code if an error occurred 18 during the call. By default, an error detected during the execution of the MPI library 19causes the parallel computation to abort, except for file operations. However, MPI provides 20mechanisms for users to change this default and to handle recoverable errors. The user 21may specify that no error is fatal, and handle error codes returned by MPI calls by himself 22or herself. Also, the user may provide his or her own error-handling routines, which will 23be invoked whenever an MPI call returns abnormally. The MPI error handling facilities 24 are described in Section 8.3. The return values of C++ functions are not error codes. 25If the default error handler has been set to $MPI::ERRORS_THROW_EXCEPTIONS$, the C++ 26exception mechanism is used to signal an error by throwing an MPI::Exception object. See 27also Section 16.1.8 on page 638.

Several factors limit the ability of MPI calls to return with meaningful error codes
 when an error occurs. MPI may not be able to detect some errors; other errors may be too
 expensive to detect in normal execution mode; finally some errors may be "catastrophic"
 and may prevent MPI from returning control to the caller in a consistent state.

32 Another subtle issue arises because of the nature of asynchronous communications: MPI 33 calls may initiate operations that continue asynchronously after the call returned. Thus, the 34operation may return with a code indicating successful completion, yet later cause an error 35 exception to be raised. If there is a subsequent call that relates to the same operation (e.g., 36 a call that verifies that an asynchronous operation has completed) then the error argument 37 associated with this call will be used to indicate the nature of the error. In a few cases, the 38 error may occur after all calls that relate to the operation have completed, so that no error 39 value can be used to indicate the nature of the error (e.g., an error on the receiver in a send 40with the ready mode). Such an error must be treated as fatal, since information cannot be 41 returned for the user to recover from it.

This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent. E.g., it is highly desirable that an erroneous receive call will not cause any part of the receiver's memory to be overwritten, beyond the area specified for receiving the message.

⁴⁷ Implementations may go beyond this document in supporting in a meaningful manner
 ⁴⁸ MPI calls that are defined here to be erroneous. For example, MPI specifies strict type

 $\mathbf{2}$

matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 8.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as write in Fortran and printf and malloc in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is independent of the action of other processes in an MPI program.

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that printf is available at the executing nodes).

```
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

The corresponding Fortran and C++ programs are also expected to complete.

An example of what is *not* required is any particular ordering of the action of these routines when called by several tasks. For example, MPI makes neither requirements nor recommendations for the output from the following program (again assuming that I/O is available at the executing nodes).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

In addition, calls that fail because of resource exhaustion or other error are not considered a violation of the requirements here (however, they are required to complete, just not to complete successfully).

2.9.2 Interaction with Signals

MPI does not specify the interaction of processes with signals and does not require that MPI be signal safe. The implementation may reserve some signals for its own use. It is required that the implementation document which signals it uses, and it is strongly recommended

that it not use SIGALRM, SIGFPE, or SIGIO. Implementations may also prohibit the use of
 MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI
 library by catching signals only on threads that do not execute MPI calls. High quality
 single-threaded implementations will be signal safe: an MPI call suspended by a signal will
 resume and complete normally after the signal is handled.

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.

Chapter 3

Point-to-Point Communication

3.1Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are send and receive. Their use is illustrated in the example below.

```
20
#include "mpi.h"
                                                                                    21
int main( int argc, char **argv )
                                                                                    22
{
                                                                                    23
  char message[20];
  int myrank;
 MPI_Status status;
 MPI_Init( &argc, &argv );
                                                                                    27
 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
                                                                                    28
                       /* code for process zero */
  if (myrank == 0)
                                                                                    29
  ſ
                                                                                    30
      strcpy(message,"Hello, there");
                                                                                    31
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
                                                                                    32
  }
                                                                                    33
  else if (myrank == 1) /* code for process one */
                                                                                    34
  {
                                                                                    35
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
                                                                                    36
      printf("received :%s:\n", message);
                                                                                    37
  }
 MPI_Finalize();
                                                                                    39
  return 0;
}
```

42In this example, process zero (myrank = 0) sends a message to process one using the send operation MPI_SEND. The operation specifies a send buffer in the sender memory 4344from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable **message** in the memory of process zero. The location, 45size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an **envelope** with the message. This envelope specifies the

7

9

1516

17

18

19

 24

25

26

38

40

41

46

47

1 message destination and contains distinguishing information that can be used by the **receive** $\mathbf{2}$ operation to select a particular message. The last three parameters of the send operation, 3 along with the rank of the sender, specify the envelope for the message sent. Process one 4 (myrank = 1) receives this message with the **receive** operation MPI_RECV. The message to 5be received is selected according to the value of its envelope, and the message data is stored 6 into the **receive buffer**. In the example above, the receive buffer consists of the storage $\overline{7}$ containing the string message in the memory of process one. The first three parameters 8 of the receive operation specify the location, size and type of the receive buffer. The next 9 three parameters are used for selecting the incoming message. The last parameter is used 10 to return information on the message just received. 11 The next sections describe the blocking send and receive operations. We discuss send,

12receive, blocking communication semantics, type matching requirements, type conversion in 13 heterogeneous environments, and more general communication modes. Nonblocking comticket262.¹⁴ munication is addressed next, followed by probing and canceling a message, channel-like ticket262.¹⁵ constructs and send-receive operations, [Nonblocking communication is addressed next, 16followed by channel-like constructs and send-receive operations,] ending with a description 17of the "dummy" process, MPI_PROC_NULL.

Blocking Send and Receive Operations 3.2

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

27	IN	buf	initial address of send buffer (choice)
28 29 30	IN	count	number of elements in send buffer (non-negative integer)
31	IN	datatype	datatype of each send buffer element (handle)
32	IN	dest	rank of destination (integer)
33 34	IN	tag	message tag (integer)
34 35	IN	comm	communicator (handle)

ticket140. 37

int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

```
ticket-248T. 39
                MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
           40
           41
```

42

43

45

36

18 19

2021

22

23 24 25

26

```
TYPE(*), DIMENSION(...), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

TYPE(MPI_Comm), INTENT(IN) :: comm

44INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
46
47
         <type> BUF(*)
48
```

INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

The blocking semantics of this call are described in Section 3.4.

3.2.2 Message Data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of **count** values, each of the type indicated by **datatype**. **count** may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 3.1.

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 4.2.

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1 MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2 and INTEGER*4, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the

Unofficial Draft for Comment Only

 $\mathbf{2}$

 31

 $45 \\ 46$

MPI datatype	C datatype
MPI_CHAR	char
	(treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char
	(treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char
	(treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	
	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
	(defined in <stddef.h>)</stddef.h>
	(treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	
Table 3.2: Predefined MPI datatypes co	orresponding to C datatypes
communication buffor this information mus	the supplied by an audicit answer
communication buffer; this information mus	
The need for such datatype information will	become clear in Section 3.3.2. (En
rationale.)	
Rationale. The datatypes MPI_C_BOOL,	, , ,
MPI_INT32_T, MPI_UINT8_T, MPI_UINT16	, , , , , , , , , , , , , , , , , , , ,

 $\mathsf{MPI_C_FLOAT_COMPLEX},\ \mathsf{MPI_C_DOUBLE_COMPLEX},\ \mathrm{and}$

MPI datatype	C datatype	Fortran datatype	1
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	2
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)	3
[ticket265.]MPI_COUNT	[ticket265.]MPI_Count	[ticket265.]INTEGER (KIND=MPI_COUNT_K	INÐ)
			5
Table 2.2. Dredsfored M	DI datatum az agungan ang	ling to both C and Fortner datatures	6
Table 5.5: Frederined M	P1 datatypes correspond	ling to both C and Fortran datatypes	7
			8
MPI_C_LONG_DOU	$BLE_COMPLEX$ have no	corresponding C++ bindings. This was	9
intentionally done to	avoid potential collision	ns with the C preprocessor and names-	10
paced C++ names.	C++ applications can u	se the C bindings with no loss of func-	11
tionality. (End of rat	tionale.)		12
			13
The datatypes MPI_AINT[and], MPI_OFFSET [], and MPI_COUNT correspond to the MPI-defined C types MPI_Aint[and], MPI_Offset[], and MPI_Count and their Fortran equivalents INTEGER (KIND=MPI_ADDRESS_KIND)[and], INTEGER (KIND= MPI_OFFSET_KIND), and INTEGER (KIND=MPI_COUNT_KIND). This is described in Ta- ble 3.3. See Section 16.3.10 for information on interlanguage communication with these types.			¹⁴ ticket: ¹⁵ ticket: ¹⁶ ticket: ¹⁷ ticket: ¹⁸ ticket: ¹⁹ ticket:
			20
.2.3 Message Envelope			21
C .	nt maggarag appressinfor	mation that can be used to distinguish	22

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the **message envelope**. These fields are

source destination tag communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 8. MPI requires that UB be no less than 32767.

The comm argument specifies the communicator that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe[:]"[]: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ..., n-1, where n is the number of

47 48

23

24

25 26

27

28

29

30

 31

32

33

34

35

36

37

38

39 40

41

44

45

46

⁴² ticket262. ⁴³ ticket262.

1processes in the group. (If the communicator is an inter-communicator, then destinations $\mathbf{2}$ are identified by their rank in the remote group. See Chapter 6.) 3 A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows com-4 munication with all processes that are accessible after MPI initialization and processes are 5identified by their rank in the group of MPI_COMM_WORLD. 6 7 Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing com-8 munication libraries, need only use the predefined variable MPI_COMM_WORLD as the 9 comm argument. This will allow communication with all the processes available at 10 initialization time. 11 12Users may define new communicators, as explained in Chapter 6. Communicators 13 provide an important encapsulation mechanism for libraries and modules. They allow 14modules to have their own disjoint communication universe and their own process 15numbering scheme. (End of advice to users.) 1617Advice to implementors. The message envelope would normally be encoded by a 18 fixed-length message header. However, the actual encoding is implementation depen-19 dent. Some of the information (e.g., source or destination) may be implicit, and need 20not be explicitly carried by messages. Also, processes may be identified by relative 21ranks, or absolute ids, etc. (End of advice to implementors.) 22 233.2.4 Blocking Receive 24The syntax of the blocking receive operation is given below. 252627MPI_RECV (buf, count, datatype, source, tag, comm, status) 28 OUT buf initial address of receive buffer (choice) 29 30 IN number of elements in receive buffer (non-negative incount 31 teger) 32 IN datatype datatype of each receive buffer element (handle) 33 rank of source or MPI_ANY_SOURCE (integer) 34 IN source 35 IN message tag or MPI_ANY_TAG (integer) tag 36 IN communicator (handle) comm 37 OUT status object (Status) status 38 39 40 int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, 41 int tag, MPI_Comm comm, MPI_Status *status) ticket-248T. 42 MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C) 43 TYPE(*), DIMENSION(..) :: buf 44 INTEGER, INTENT(IN) :: count, source, tag 45TYPE(MPI_Datatype), INTENT(IN) :: datatype 46 TYPE(MPI_Comm), INTENT(IN) :: comm 47TYPE(MPI_Status) :: status 48

	1		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2		
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)			
<type> BUF(*)</type>	3 4		
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	5		
IERROR	6		
<pre>{void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,</pre>	7		
int source, int tag, MPI::Status& status) const(binding	8		
deprecated, see Section 15.2) }	9		
<pre>{void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,</pre>	10		
int source, int tag) const(binding deprecated, see Section 15.2) }	11 12		
The blocking semantics of this call are described in Section 3.4.	13		
The receive buffer consists of the storage containing count consecutive elements of the	14		
type specified by datatype, starting at address buf. The length of the received message must	15		
be less than or equal to the length of the receive buffer. An overflow error occurs if all	16 17		
incoming data does not fit, without truncation, into the receive buffer.	18		
If a message that is shorter than the receive buffer arrives, then only those locations	19		
corresponding to the (shorter) message are modified.	20		
Advice to users. The MPI_PROBE function described in Section 3.8 can be used to	21		
receive messages of unknown length. (End of advice to users.)	22		
	23		
Advice to implementors. Even though no specific behavior is mandated by MPI for	24		
erroneous programs, the recommended handling of overflow situations is to return in	25		
status information about the source and tag of the incoming message. The receive	26		
operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.	27		
•	28 29		
In the case of a message shorter than the receive buffer, MPI is quite strict in that it	30		
allows no modification of the other locations. A more lenient statement would allow	31		
for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if	32		
it is an odd address. (End of advice to implementors.)	33		
To is an old address. (<i>Lina of dance to implementors.</i>)	34		
The selection of a message by a receive operation is governed by the value of the	35		
nessage envelope. A message can be received by a receive operation if its envelope matches ³⁶ he source, tag and comm values specified by the receive operation. The receiver may ³⁷			
value for tag, indicating that any source and/or tag are acceptable. It cannot specify a	39		
wildcard value for comm . Thus, a message can be received by a receive operation only if it is addressed to the receiving process has a matching communicator has matching	40 41		
α α β β α β	**		

tag=MPI_ANY_TAG in the pattern. The message tag is specified by the tag argument of the receive operation. 44The argument source, if different from MPI_ANY_SOURCE, is specified as a rank within the 4546process group associated with that same communicator (remote process group, for in-47tercommunicators). Thus, the range of valid values for the source argument is $\{0, ..., n-1\}$ 48 $1 \cup \{ MPI_ANY_SOURCE \}, where n is the number of processes in this group.$

if it is addressed to the receiving process, has a matching communicator, has matching

source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless

Unofficial Draft for Comment Only

42

1 Note the asymmetry between send and receive operations: A receive operation may $\mathbf{2}$ accept messages from an arbitrary sender, on the other hand, a send operation must specify 3 a unique receiver. This matches a "push" communication mechanism, where data transfer 4 is effected by the sender (rather than a "pull" mechanism, where data transfer is effected 5by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 3.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (End of advice to *implementors.*)

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPIdefined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus,

25status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and 26error code, respectively, of the received message. 27

In Fortran with USE mpi or INCLUDE 'mpif.h', status is an array of INTEGERs of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

With Fortran USE mpi_f08, status is defined as the Fortran BIND(C) derived type TYPE(MPI_Status) containing three public fields named MPI_SOURCE,

MPI_TAG, and MPI_ERROR. TYPE(MPI_Status) may contain additional, implementationspecific fields. Thus, status%MPI_SOURCE, status%MPI_TAG and status%MPI_ERROR contain the source, tag, and error code of a received message respectively. Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined to allow conversion between both status representations. Conversion routines are provided in Section 16.3.5 on page 698.

- Rationale. The Fortran TYPE(MPI_Status) is defined as a BIND(C) derived type so that it can be used at any location where the status integer array representation can be used, e.g., in user defined common blocks. (End of rationale.)
- It is allowed to have the same name (e.g., MPI_SOURCE) defined as a Rationale. constant (e.g., Fortran parameter) and as a field of a derived type. (End of rationale.)
- In C++, the status object is handled through the following methods:

Unofficial Draft for Comment Only

6

 $\overline{7}$

8

9

10

11

12

13

141516

17

18

19

20

21

22

23

 24

29

30

31

33

34

35

36

37

38

39

40 41

42

43

4445

46

47 48

ticket243-O. 28

```
ticket243-O. 32
```

{int MPI::Status::Get_source() const(binding deprecated, see Section 15.2) }
{void MPI::Status::Set_source(int source)(binding deprecated, see Section 15.2) }
{int MPI::Status::Get_tag() const(binding deprecated, see Section 15.2) }
{void MPI::Status::Set_tag(int tag)(binding deprecated, see Section 15.2) }
{int MPI::Status::Get_error() const(binding deprecated, see Section 15.2) }
{void MPI::Status::Set_error(int error)(binding deprecated, see Section 15.2) }

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI_ERR_IN_STATUS.

Rationale. The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to "decode" this information.

MPI_GET_COUNT(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype of each receive buffer entry (handle)
OUT	count	number of received entries (integer)

	$_{34}^{35}$ ticket-248T.	
<pre>MPI_Get_count(status, datatype, count, ierror) BIND(C) TYPE(MPI_Status), INTENT(IN) :: status</pre>	35	
	36	
TYPE(MPI_Datatype), INTENT(IN) :: datatype	37	
INTEGER, INTENT(OUT) :: count	38	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39	
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)	40	
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	41	
(int MDT, Otstand, Ost sound (sound MDT, Detation of Jatations) sound (his Jins	42	
{int MPI::Status::Get_count(const MPI::Datatype& datatype) const(binding		
deprecated, see Section 15.2) }	44	
Returns the number of entries received. (Again, we count <i>entries</i> , each of type <i>datatype</i> ,	45	

Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, not *bytes*.) The **datatype** argument should match the argument provided by the receive call that set the **status** variable. If the number of entries received exceeds the limits of the count parameter, then MPI_GET_COUNT sets the value of count to MPI_UNDEFINED. [(We

⁴⁷ ticket265. ⁴⁸ ticket265.

Unofficial Draft for Comment Only

 $\mathbf{2}$

³² ticket140.

shall later see, in Section 4.1.11, that MPI_GET_COUNT may return, in certain situations, the value MPI_UNDEFINED.)]There are other situations where the value of count can be set to MPI_UNDEFINED; see Section 4.1.11.

Rationale. Some message-passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the "last message received." This is not thread safe.

The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE or MPI_IPROBE. With a status from MPI_PROBE or MPI_IPROBE, the same datatypes are allowed as in a call to MPI_RECV to receive this message. (*End of rationale.*)

The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transfered is greater than zero, MPI_UNDEFINED is returned.

Rationale. Zero-length datatypes may be created in a number of cases. An important case is MPI_TYPE_CREATE_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI_GET_COUNT to check the status. (*End of rationale.*)

Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI_GET_COUNT and the receive. (*End of advice to users.*)

All send and receive operations use the buf, count, datatype, source, dest, tag, comm and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

36 37 3.2.6 Passing MPI_STATUS_IGNORE for Status

³⁸ Every call to MPI_RECV includes a status argument, wherein the system can return details ³⁹ about the message received. There are also a number of other MPI calls where status ⁴⁰ is returned. An object of type MPI_STATUS is not an MPI opaque object; its structure ⁴¹ is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, ⁴² application programs are constructed so that it is unnecessary for them to examine the ⁴³ status fields. In these cases, it is a waste for the user to allocate a status object, and it is ⁴⁴ particularly wasteful for the MPI implementation to fill in fields in this object.

ticket229.2. 46

⁴⁵ To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE ⁴⁶ and MPI_STATUSES_IGNORE, which when passed to a receive, probe, wait, or test function, ⁴⁷ inform the implementation that the status fields are not to be filled in. Note that

⁴⁸ MPI_STATUS_IGNORE is not a special type of MPI_STATUS object; rather, it is a special

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

21

22

23

24

25

26 27

28

29

30

31 32

33

34

1 value for the argument. In C one would expect it to be NULL, not the address of a special $\mathbf{2}$ MPI_STATUS. 3 MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used every-4 where a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE 5and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or 6 $\overline{7}$ assignment). See Section 2.5.4. In general, this optimization can apply to all functions for which status or an array of 8 9 statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV, 10 MPI_PROBE, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When ¹¹ ticket229.2. an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, 1213 MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE 1415has been passed to that function. 16MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are not required to have the same 17 values in C and Fortran. 18 It is not allowed to have some of the statuses in an array of statuses for MPI_{TEST|WAIT}{ALL|SOME} functions set to MPI_STATUS_IGNORE; one either specifies 1920ignoring all of the statuses in such a call with MPI_STATUSES_IGNORE, or *none* of them by passing normal statuses in all positions in the array of statuses. 21 ticket262. In the deprecated C++ bindings, there There are no C++ bindings for 22 MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE. To allow an OUT or INOUT MPI::Status 23 24 argument to be ignored, all MPI C++ bindings that have OUT or INOUT MPI::Status 25parameters are overloaded with a second version that omits the OUT or INOUT MPI::Status 26parameter. 27**Example 3.1** The [deprecated C++ bindings for MPI_PROBE are: 28 ticket 262. void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const 29 void MPI::Comm::Probe(int source, int tag) const 30 31 32 3.3 Data Type Matching and Data Conversion 33 34 3.3.1 Type Matching Rules 35One can think of message transfer as consisting of the following three phases. 36 37 1. Data is pulled out of the send buffer and a message is assembled. 38 39 2. A message is transferred from sender to receiver. 40 3. Data is pulled from the incoming message and disassembled into the receive buffer. 41 42Type matching has to be observed at each of these three phases: The type of each 43 variable in the sender buffer has to match the type specified for that entry by the send 44operation; the type specified by the send operation has to match the type specified by the 45receive operation; and the type of each variable in the receive buffer has to match the type 46specified for that entry by the receive operation. A program that fails to observe these three 47rules is erroneous.

¹ To define type matching more precisely, we need to deal with two issues: matching of ² types of the host language with types specified in communication operations; and matching ³ of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL,
 and so on. There is one exception to this rule, discussed in Section 4.2, the type
 MPI_PACKED can match any other type.

8 The type of a variable in a host program matches the type specified in the commu-9 nication operation if the datatype name used by that operation corresponds to the basic 10 type of the host program variable. For example, an entry with type name MPI_INTEGER 11matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran 12and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with 13 type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a 14byte-addressable machine), irrespective of the datatype of the variable that contains this 15byte. The type MPI_PACKED is used to send data that has been explicitly packed, or 16receive data that will be explicitly unpacked, see Section 4.2. The type MPI_BYTE allows 17one to transfer the binary value of a byte in memory unchanged.

18

19

20

21

22 23

24

25

26

27

28 29

30 31

32

To summarize, the type matching rules fall into the three categories below.

• Communication of typed values (e.g., with datatype different from MPI_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.

- Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender and receiver use the datatype MPI_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI_PACKED is used.

The following examples illustrate the first two cases.

Example 3.2 Sender and receiver specify matching types.

```
33
     CALL MPI_COMM_RANK(comm, rank, ierr)
34
     IF (rank.EQ.0) THEN
35
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
36
     ELSE IF (rank.EQ.1) THEN
37
          CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
38
     END IF
39
          This code is correct if both a and b are real arrays of size \geq 10. (In Fortran, it might
40
     be correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
41
```

```
to an array with ten reals.)
```

- ⁴⁴ **Example 3.3** Sender and receiver do not specify matching types.

CALL MPI_COMM_RANK(comm, rank, ierr)					
IF (rank.EQ.0) THEN					
CALL MPI_SEND(a(1), 10, MPI_REAL,	1, tag, comm, ierr)				
ELSE IF (rank.EQ.1) THEN					
CALL MPI_RECV(b(1), 40, MPI_BYTE,	0, tag, comm, status, ierr)				
END IF					

This code is erroneous, since sender and receiver do not provide matching datatype arguments.

Example 3.4 Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of a and b (unless this results in an out of bound memory access).

Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

Type MPI_CHARACTER

The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather then the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

```
37
Example 3.5
                                                                                       38
    Transfer of Fortran CHARACTERs.
                                                                                       39
                                                                                       40
CHARACTER*10 a
                                                                                       41
CHARACTER*10 b
                                                                                       42
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                       43
                                                                                       44
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
                                                                                       45
                                                                                       46
ELSE IF (rank.EQ.1) THEN
                                                                                       47
    CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
                                                                                       48
END IF
```

Unofficial Draft for Comment Only

The last five characters of string **b** at process 1 are replaced by the first five characters of string **a** at process 0.

Rationale. The alternative choice would be for MPI_CHARACTER to match a character of arbitrary length. This runs into problems.

6 A Fortran character variable is a constant length string, with no special termina-7 tion symbol. There is no fixed convention on how to represent characters, and how 8 to store their length. Some compilers pass a character argument to a routine as a 9 pair of arguments, one holding the address of the string and the other holding the 10 length of string. Consider the case of an MPI communication call that is passed a 11 communication buffer with type defined by a derived datatype (Section 4.1). If this 12communicator buffer contains variables of type CHARACTER then the information on 13 their length will not be passed to the MPI routine. 14

This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (*End of rationale.*)

Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (End of advice to implementors.)

²⁴ 3.3.2 Data Conversion

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. We
 use the following terminology.

³⁰ type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.

representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can be represented on the target system.

⁴⁰ Overflow and underflow exceptions may occur during floating point conversions. Con-⁴¹ version of integers or characters may also lead to exceptions when a value that can be ⁴² represented in one system cannot be represented in the other system. An exception occur-⁴³ ring during representation conversion results in a failure of the communication. An error ⁴⁴ occurs either in the send operation, or the receive operation, or both.

⁴⁵ If a value sent in a message is untyped (i.e., of type MPI_BYTE), then the binary ⁴⁶ representation of the byte stored at the receiver is identical to the binary representation ⁴⁷ of the byte loaded at the sender. This holds true, whether sender and receiver run in the ⁴⁸ same or in distinct environments. No representation conversion is required. (Note that

1

2

3 4

5

19

20

21

22

23

29

 31

32

representation conversion may occur when values of type MPI_CHARACTER or MPI_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

Consider the three examples, 3.2-3.4. The first program is correct, assuming that a and b are REAL arrays of size ≥ 10 . If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

The second program is erroneous, and its behavior is undefined.

The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver run in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If **a** and **b** are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)

MPI requires support for inter-language communication, i.e., if messages are sent by a C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined in Section 16.3 on page 694.

3.4 Communication Modes

The send call described in Section 3.2.1 is **blocking**: it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI offers the choice of several communication modes that allow one to control the choice of the communication protocol.

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12 13

14 15

16

17

18

19

20

21

22 23

 24

25

26

27

28 29

30

31

32

33

34

35 36 37

38 39

40 41

The send call described in Section 3.2.1 uses the **standard** communication mode. In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

⁸ Thus, a send in standard mode can be started whether or not a matching receive has ⁹ been posted. It may complete before a matching receive is posted. The standard mode send ¹⁰ is **non-local**: successful completion of the send operation may depend on the occurrence ¹¹ of a matching receive.

Rationale. The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. If the user wishes to guarantee a certain amount of buffering, the userprovided buffer system of Section 3.6 should be used, along with the buffered-mode send. (End of rationale.)

21 22 23

12

13

14

15 16

17

18 19

20

There are three additional communication modes.

A **buffered** mode send operation can be started whether or not a matching receive 24 has been posted. It may complete before a matching receive is posted. However, unlike 25the standard send, this operation is **local**, and its completion does not depend on the 26occurrence of a matching receive. Thus, if a send is executed and no matching receive is 27posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. 28An error will occur if there is insufficient buffer space. The amount of available buffer space 29 is controlled by the user — see Section 3.6. Buffer allocation by the user may be required 30 for the buffered mode to be effective. 31

A send that uses the **synchronous** mode can be started whether or not a matching 32 receive was posted. However, the send will complete successfully only if a matching receive is 33 posted, and the receive operation has started to receive the message sent by the synchronous 34send. Thus, the completion of a synchronous send not only indicates that the send buffer 35 can be reused, but it also indicates that the receiver has reached a certain point in its 36 execution, namely that it has started executing the matching receive. If both sends and 37 receives are blocking operations then the use of the synchronous mode provides synchronous 38 communication semantics: a communication does not complete at either end before both 39 processes rendezvous at the communication. A send executed in this mode is **non-local**. 40

A send that uses the **ready** communication mode may be started *only* if the matching 41 receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-42fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 43 required and results in improved performance. The completion of the send operation does 44not depend on the status of a matching receive, and merely indicates that the send buffer 45can be reused. A send operation that uses the ready mode has the same semantics as a 46standard send operation, or a synchronous send operation; it is merely that the sender 47provides additional information to the system (namely that a matching receive is already 48

posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)	10
IN	count	number of elements in send buffer (non-negative inte-	11
		ger)	12
		0. /	13
IN	datatype	datatype of each send buffer element (handle)	14
IN	dest	rank of destination (integer)	15
			16
IN	tag	message tag (integer)	17
			17
IN	comm	communicator (handle)	18

int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest, 20 ticket 140. int tag, MPI_Comm comm) 21₂₂ ticket-248T. MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C) 23TYPE(*), DIMENSION(...), INTENT(IN) :: buf 24 INTEGER, INTENT(IN) :: count, dest, tag 25TYPE(MPI_Datatype), INTENT(IN) :: datatype 26TYPE(MPI_Comm), INTENT(IN) :: comm 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 29 30 <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 3132 {void MPI::Comm::Bsend(const void* buf, int count, const 33 MPI::Datatype& datatype, int dest, int tag) const(binding

Send in buffered mode.

deprecated, see Section 15.2 }

1

2

3

4

56

7 8

9

19

34

35

1 MPI_SSEND (buf, count, datatype, dest, tag, comm) $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN count number of elements in send buffer (non-negative inte-4 ger) 56 IN datatype of each send buffer element (handle) datatype 7 IN dest rank of destination (integer) 8 IN tag message tag (integer) 9 10IN comm communicator (handle) 11 12ticket140. int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest, 13 int tag, MPI_Comm comm) ticket-248T. 14 MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) BIND(C) 15TYPE(*), DIMENSION(...), INTENT(IN) :: buf 16INTEGER, INTENT(IN) :: count, dest, tag 17TYPE(MPI_Datatype), INTENT(IN) :: datatype 18 TYPE(MPI_Comm), INTENT(IN) :: comm 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 22 <type> BUF(*) 23INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 24{void MPI::Comm::Ssend(const void* buf, int count, const 2526MPI::Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2 } 2728Send in synchronous mode. 29 30 31 MPI_RSEND (buf, count, datatype, dest, tag, comm) 32 IN buf initial address of send buffer (choice) 33 IN count number of elements in send buffer (non-negative inte-34 ger) 35 36 IN datatype datatype of each send buffer element (handle) 37 IN dest rank of destination (integer) 38 IN message tag (integer) 39 tag 40 IN communicator (handle) comm 41 42ticket140. int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest, 43 int tag, MPI_Comm comm) ticket-248T. 44 MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C) 45TYPE(*), DIMENSION(...), INTENT(IN) :: buf 46 INTEGER, INTENT(IN) :: count, dest, tag 47TYPE(MPI_Datatype), INTENT(IN) :: datatype 48

TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	3 4
<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR</type>	5 6
<pre>{void MPI::Comm::Rsend(const void* buf, int count, const</pre>	7
<pre>MPI::Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>	8 9 10
Send in ready mode. There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is blocking : it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started). In a multi-threaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.	11 12 14 14 15 14 15 16 17 18 19 20
Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.	21 22 23
It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.	24 25 26
A possible communication protocol for the various communication modes is outlined below.	27 28
ready send: The message is sent as soon as possible.	29 30
synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.	31 32 33
standard send : First protocol may be used for short messages, and second protocol for long messages.	34 35
buffered send : The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).	36 37 38
Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.	39 40
Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.	41 42
A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.	43 44 45
In a multi-threaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (<i>End of advice to implementors.</i>)	46 47 48

3.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

6 **Order** Messages are *non-overtaking*: If a sender sends two messages in succession to the 7same destination, and both match the same receive, then this operation cannot receive the 8 second message if the first one is still pending. If a receiver posts two receives in succession, 9 and both match the same message, then the second receive operation cannot be satisfied 10 by this message, if the first one is still pending. This requirement facilitates matching of 11sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the 12calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of 13 14nondeterminism.)

If a process has a single thread of execution, then any two communications executed 1516by this process are ordered. On the other hand, if the process is multi-threaded, then the 17semantics of thread execution may not define a relative order between two send operations 18 executed by two distinct threads. The operations are logically concurrent, even if one 19physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two 2021successively sent messages, then the two messages can match the two receives in either 22order.

Example 3.6 An example of non-overtaking messages. 24

```
CALL MPI_COMM_RANK(comm, rank, ierr)
26
     IF (rank.EQ.0) THEN
27
         CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
28
         CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
29
    ELSE IF (rank.EQ.1) THEN
30
         CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
^{31}
         CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
32
    END IF
33
```

34The message sent by the first send must be received by the first receive, and the message 35 sent by the second send must be received by the second receive. 36

Progress If a pair of matching send and receives have been initiated on two processes, then 38 at least one of these two operations will complete, independently of other actions in the 39 system: the send operation will complete, unless the receive is satisfied by another message, 40 and completes; the receive operation will complete, unless the message sent is consumed by 41 another matching receive that was posted at the same destination process. 42

43 Example 3.7 An example of two, intertwined matching pairs. 44

4546

37

- 47
- 48

1 $\mathbf{2}$

3

4 5

23

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multi-threaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI call. A quality implementation will use a (small) fixed amount of resources for each pending send in the ready or synchronous mode and for each pending receive. However, buffer space may be consumed to store messages sent in standard mode, and must be consumed to store messages sent in buffered mode, when no matching receive is available. The amount of space available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

MPI allows the user to provide buffer memory for messages sent in the buffered mode. Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI implementation is required to do no worse than implied by this model. This allows users to avoid buffer overflows when they use buffered sends. Buffer allocation and use is described in Section 3.6.

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signalled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow

Unofficial Draft for Comment Only

 24

 31

1 will result. Additional synchronization has to be added to the program so as to prevent $\mathbf{2}$ this from occurring. If standard sends are used, then the producer will be automatically 3 throttled, as its send operations will block when buffer space is unavailable. 4 In some situations, a lack of buffer space leads to deadlock situations. This is illustrated $\mathbf{5}$ by the examples below. 6 Example 3.8 An exchange of messages. 78 CALL MPI_COMM_RANK(comm, rank, ierr) 9 IF (rank.EQ.0) THEN 10 CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) 11 CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) 12ELSE IF (rank.EQ.1) THEN 13 CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) 14CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) 15END IF 1617This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send. 1819An errant attempt to exchange messages. Example 3.9 2021CALL MPI_COMM_RANK(comm, rank, ierr) 22IF (rank.EQ.0) THEN 23CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) 24 CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) 25ELSE IF (rank.EQ.1) THEN 26CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) 27CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) 28END IF 29The receive operation of the first process must complete before its send, and can complete 30 only if the matching send of the second processor is executed. The receive operation of the 31 second process must complete before its send and can complete only if the matching send 32 of the first process is executed. This program will always deadlock. The same holds for any 33 other send mode. 34 35Example 3.10 An exchange that relies on buffering. 36 CALL MPI_COMM_RANK(comm, rank, ierr) 37 IF (rank.EQ.0) THEN 38 CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr) 39 CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr) 40 ELSE IF (rank.EQ.1) THEN 41 CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr) 42CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr) 43 END IF 4445The message sent by each process has to be copied out before the send operation returns 46and the receive operation starts. For the program to complete, it is necessary that at least 47one of the two messages sent be buffered. Thus, this program can succeed only if the

⁴⁸ communication system can buffer at least **count** words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

A program is "safe" if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.10. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

3.6 Buffer Allocation and Usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

33 MPI_BUFFER_ATTACH(buffer, size) 34 IN buffer initial buffer address (choice) 35IN size buffer size, in bytes (non-negative integer) 36 37 int MPI_Buffer_attach(void* buffer, int size) 38 39 ticket-248T. MPI_Buffer_attach(buffer, size, ierror) BIND(C) 40 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer 41 INTEGER, INTENT(IN) :: size 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR) 44 <type> BUFFER(*) 4546 INTEGER SIZE, IERROR 4748

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16

17

18

19

20 21

22

23 24

25

26 27

28 29

30

		50		CHAPTER 3.	POINT-TO-POINT COMMUNICATION	
	1 2	{void MPI	::Attach_buffer(v Section 15.2) }		nt size)(binding deprecated, see	
ticket229.2.	memory region. In Fortran, one can pass the first element of a memory region or a who array, which must be 'simply contiguous' (for 'simply contiguous', see also Section 16.2.1 on page 673).					
	10 11	MPI BUFF	ER_DETACH(buffer_	addr size)		
	12	OUT	buffer_addr	,	buffer address (choice)	
	13 14	OUT	size		size, in bytes (non-negative integer)	
	15					
ticket229.2.	16	int MPI_E	Buffer_detach(void	* buffer_addr,	int* size)	
ticket-248T.		MPI_Buffe	er_detach(buffer_a	ddr, size, ier	ror) BIND(C)	
	19		INTRINSIC :: ISC			
	20		(C_PTR), INTENT(OU		addr	
	21		ER, INTENT(OUT) : ER, OPTIONAL, INT		error	
	22 23					
	24		ER_DETACH (BUFFER_A	DDR, SIZE, IER	ROR)	
	25	• -	<pre>> BUFFER_ADDR(*) GER SIZE, IERROR</pre>			
	26			idue buffan) (b	inding dominanted and Castian 15 0)	
	27 28			· ·	inding deprecated, see Section 15.2) }	
	28 29				MPI. The call returns the address and the	
	30				l block until all messages currently in the s function, the user may reuse or deallocate	
	31		taken by the buffer.	poin return or tim	s function, the user may reuse of deanocate	
	32	-	,			
	33 34	Example	3.11 Calls to attac	ch and detach but	ffers.	
	35	#define E	BUFFSIZE 10000			
	36	int size;				
	37	char *buf		·		
	38		er_attach(malloc(
	39		er of 10000 bytes er_detach(&buff,		ed by MPI_Bsend */	
	40 41		size reduced to			
	41		er_attach(buff, s			
	43		of 10000 bytes a		. */	
	44		_	-		
	45			0	nctions MPI_Buffer_attach and	
	46			-	ent of type void*, these arguments are used ed to MPI_Buffer_attach; the address of the	
	47		· .	-	that this call can return the pointer value.	
	48	point	TO PRODUCE UP INIT _		the one can can recurrence pointer value.	

ticket229.2.

In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the mpi_f08 module, the address of the buffer is returned as TYPE(C_PTR), see also Example 8.1 on page 355 about the use of C_PTR pointers. (*End of advice to users.*)

Rationale. Both arguments are defined to be of type void* (rather than void* and void**, respectively), so as to avoid complex type casts. E.g., in the last example, &buff, which is of type char**, can be passed as argument to MPI_Buffer_detach without type casting. If the formal parameter had type void** then we would need a type cast before and after the call. (*End of rationale.*)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages *as if* outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

Rationale. There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 4.2 and the nonblocking communication functions described in Section 3.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffered send call results in the execution of the following code.

• Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 31

 $45 \\ 46$

• Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function MPI_PACK_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).

• Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.

- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return
- 17 18 19

20

3.7 Nonblocking Communication

21One can improve performance on many systems by overlapping communication and com-22 putation. This is especially true on systems where communication can be executed au-23tonomously by an intelligent communication controller. Light-weight threads are one mech- 24 anism for achieving such overlap. An alternative mechanism that often leads to better 25performance is to use **nonblocking communication**. A nonblocking **send start** call ini-26tiates the send operation, but does not complete it. The send start call can return before 27the message was copied out of the send buffer. A separate send complete call is needed 28 to complete the communication, i.e., to verify that the data has been copied out of the send 29buffer. With suitable hardware, the transfer of data out of the sender memory may proceed 30 concurrently with computations done at the sender after the send was initiated and before it 31 completed. Similarly, a nonblocking **receive start call** initiates the receive operation, but 32 does not complete it. The call can return before a message is stored into the receive buffer. 33 A separate **receive complete** call is needed to complete the receive operation and verify 34that the data has been received into the receive buffer. With suitable hardware, the transfer 35 of data into the receiver memory may proceed concurrently with computations done after 36 the receive was initiated and before it completed. The use of nonblocking receives may also 37 avoid system buffering and memory-to-memory copying, as information is provided early 38 on the location of the receive buffer.

39 Nonblocking send start calls can use the same four modes as blocking sends: standard, 40 buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready 41 excepted, can be started whether a matching receive has been posted or not; a nonblocking 42ready send can be started only if a matching receive is posted. In all cases, the send start call 43 is local: it returns immediately, irrespective of the status of other processes. If the call causes 44some system resource to be exhausted, then it will fail and return an error code. Quality 45implementations of MPI should ensure that this happens only in "pathological" cases. That 46is, an MPI implementation should be able to support a large number of pending nonblocking 47operations. 48

1

2

3

4

5

6

7

8

9

10

11 12

13

14 15

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is synchronous, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender "knows" the transfer will complete, but before the receiver "knows" the transfer will complete.)

If the send mode is **buffered** then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

If the send mode is **standard** then the send-complete call may return before a matching receive is posted, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Advice to users. The completion of a send operation may be delayed, for standard mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., the blocking version of buffered send is capable of completing regardless of when a matching receive call is made. However, separating the start from the completion of these sends still gives some opportunity for optimization within the MPI library. For example, starting a buffered send gives an implementation more flexibility in determining if and how the message is buffered. There are also advantages for both nonblocking buffered and ready modes when data copying can be done concurrently with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (*End of advice to users.*)

3.7.1 Communication Request Objects

Nonblocking communications use opaque **request** objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38 39

40 41

42

43

44

45

46

47

```
1
                  3.7.2
                         Communication Initiation
            \mathbf{2}
                  We use the same naming conventions as for blocking communication: a prefix of B, S, or
            3
                  R is used for buffered, synchronous or ready mode. In addition a prefix of I (for immediate)
            4
                  indicates that the call is nonblocking.
            5
            6
            \overline{7}
                  MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
            8
                              buf
                    IN
                                                          initial address of send buffer (choice)
            9
            10
                    IN
                                                          number of elements in send buffer (non-negative inte-
                             count
            11
                                                           ger)
            12
                    IN
                             datatype
                                                          datatype of each send buffer element (handle)
            13
                    IN
                              dest
                                                          rank of destination (integer)
            14
            15
                    IN
                             tag
                                                          message tag (integer)
            16
                    IN
                                                          communicator (handle)
                              comm
            17
                    OUT
                                                          communication request (handle)
                              request
            18
            19
  ticket140. 20
                  int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,
                                 int tag, MPI_Comm comm, MPI_Request *request)
            21
ticket-248T. _{22}^{-1}
                  MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
            23
                      TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
            24
                      INTEGER, INTENT(IN) :: count, dest, tag
            25
                      TYPE(MPI_Datatype), INTENT(IN) :: datatype
            26
                      TYPE(MPI_Comm), INTENT(IN) :: comm
            27
                      TYPE(MPI_Request), INTENT(OUT) :: request
            28
                      INTEGER, OPTIONAL, INTENT(OUT) ::
                                                               ierror
            29
                  MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
            30
                      <type> BUF(*)
            ^{31}
                      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
            32
            33
                  {MPI::Request MPI::Comm::Isend(const void* buf, int count, const
            34
                                 MPI::Datatype& datatype, int dest, int tag) const/binding
            35
                                 deprecated, see Section 15.2 }
            36
                      Start a standard mode, nonblocking send.
            37
            38
            39
            40
            41
            42
            43
            44
            45
            46
            47
            48
```

54

CHAPTER 3. POINT-TO-POINT COMMUNICATION

3.7. NONBLOCKING COMMUNICATION

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)					
IN	buf	initial address of send buffer (choice)	2 3		
IN	count	number of elements in send buffer (non-negative inte-	4		
		ger)	5		
IN	datatype	datatype of each send buffer element (handle)	6		
IN	dest	rank of destination (integer)	7		
IN	tag	message tag (integer)	8		
IN	comm	communicator (handle)	10		
			11		
OUT	request	communication request (handle)	12		
int MPT T	beend (const woid* buf i	nt count, MPI_Datatype datatype, int dest,	$^{13}_{14}$ ticket 140.		
IIIC MII_I		<pre>mm, MPI_Request *request)</pre>	15		
MDT Theor	0		$^{15}_{16}$ ticket-248T.		
	• -	<pre>dest, tag, comm, request, ierror) BIND(C) T(IN), ASYNCHRONOUS :: buf</pre>	17		
	ER, INTENT(IN) :: count		18		
	MPI_Datatype), INTENT(IN)	-	19		
	MPI_Comm), INTENT(IN) ::		20		
TYPE(MPI_Request), INTENT(OUT)) :: request	21		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)					
	> BUF(*)	JESI, IAG, COMM, REQUESI, IERROR)	24		
01		, TAG, COMM, REQUEST, IERROR	25 26		
MDTPog	uset MDT. Comm. Theond (a	onst void* buf, int count, const	27		
Jurtuned	•	ype, int dest, int tag) const(binding	28		
	deprecated, see Section 1		29		
Ct			30		
Start	a buffered mode, nonblocking	sena.	31		
			32		
MPI_ISSEN	D(buf, count, datatype, dest,	tag, comm, request)	33 34		
IN	buf	initial address of send buffer (choice)	35		
IN	count	number of elements in send buffer (non-negative inte-	36		
	count	ger)	37		
IN	datatype	datatype of each send buffer element (handle)	38		
		· -	39		
IN	dest	rank of destination (integer)	40		
IN	tag	message tag (integer)	41 42		
IN	comm	communicator (handle)	43		
OUT	request	communication request (handle)	44		
			45		
<pre>int MPI_Issend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>					

Unofficial Draft for Comment Only

 $_{\rm 48}$ ticket-248T.

```
1
                 MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
            \mathbf{2}
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
            3
                     INTEGER, INTENT(IN) :: count, dest, tag
            4
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
            5
                     TYPE(MPI_Comm), INTENT(IN) :: comm
            6
                     TYPE(MPI_Request), INTENT(OUT) :: request
            7
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            8
                 MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
            9
                     <type> BUF(*)
           10
                     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
           11
           12
                 {MPI::Request MPI::Comm::Issend(const void* buf, int count, const
           13
                               MPI::Datatype& datatype, int dest, int tag) const(binding
           14
                                deprecated, see Section 15.2 }
           15
                     Start a synchronous mode, nonblocking send.
           16
           17
           18
                 MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)
           19
                   IN
                            buf
                                                        initial address of send buffer (choice)
           20
           21
                   IN
                            count
                                                        number of elements in send buffer (non-negative inte-
           22
                                                        ger)
           23
                   IN
                            datatype
                                                        datatype of each send buffer element (handle)
           24
                   IN
                            dest
                                                        rank of destination (integer)
           25
           26
                   IN
                                                        message tag (integer)
                            tag
           27
                   IN
                            comm
                                                        communicator (handle)
           28
                   OUT
                            request
                                                        communication request (handle)
           29
           30
  ticket140. 31
                 int MPI_Irsend(const void* buf, int count, MPI_Datatype datatype, int dest,
                                int tag, MPI_Comm comm, MPI_Request *request)
           32
ticket-248
T<br/>._{\scriptscriptstyle 33}
                 MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
           34
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
           35
                     INTEGER, INTENT(IN) :: count, dest, tag
           36
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           37
                     TYPE(MPI_Comm), INTENT(IN) :: comm
           38
                     TYPE(MPI_Request), INTENT(OUT) :: request
           39
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           40
                 MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
           41
           42
                     <type> BUF(*)
                     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
           43
           44
                 {MPI::Request MPI::Comm::Irsend(const void* buf, int count, const
           45
                               MPI::Datatype& datatype, int dest, int tag) const/binding
           46
                                deprecated, see Section 15.2 }
           47
           48
                     Start a ready mode nonblocking send.
```

			1		
MPI_IREC	V (buf, count, datatype, source	, tag, comm, request)	1 2		
OUT	buf	initial address of receive buffer (choice)	3		
IN	count	number of elements in receive buffer (non-negative in-	4		
		teger)	5		
IN	datatype	datatype of each receive buffer element (handle)	6		
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7		
			8		
IN	tag	message tag or MPI_ANY_TAG (integer)	9		
IN	comm	communicator (handle)	10		
OUT	request	communication request (handle)	11 12		
			12		
int MPI_1	<pre>Irecv(void* buf, int count</pre>	t, MPI_Datatype datatype, int source,	14		
	int tag, MPI_Comm co	mm, MPI_Request *request)	¹⁵ ticket-248T.		
MPI Irecu	(buf. count. datatype. so	<pre>purce, tag, comm, request, ierror) BIND(C)</pre>	16 ticket-2461.		
	(*), DIMENSION(), ASYNCH	· · · · · · · · · · · · · · · · · · ·	17		
INTEGER, INTENT(IN) :: count, source, tag					
TYPE(MPI_Datatype), INTENT(IN) :: datatype					
TYPE	(MPI_Comm), INTENT(IN) ::	comm	20		
	(MPI_Request), INTENT(OUT)	-	21		
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	22 23		
MPI_IRECV	(BUF, COUNT, DATATYPE, SO	DURCE, TAG, COMM, REQUEST, IERROR)	23		
<type> BUF(*)</type>					
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR					
∫MPT··Roc	uest MPI::Comm::Irecv(voi	id the int count const	27		
	-	ype, int source, int tag) const (binding	28		
	deprecated, see Section		29		
CL I		/)	30		
	a nonblocking receive.	an accurate abject and according to it with the accurat	31		
		on request object and associate it with the request quest can be used later to query the status of the	32		
	ation or wait for its completio		33		
		hat the system may start copying data out of the	34 35		
	8	ify any part of the send buffer after a nonblocking	36		
	ation is called, until the send of		37		
	0	that the system may start writing data into the re-	38		
ceive buffer. The receiver should not access any part of the receive buffer after a nonblocking					

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in [subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.10 on pages 673 and 679.]Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 673-676 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 16.2.16 to 16.2.19 on pages 679 to 688 about "Optimization Problems",

receive operation is called, until the receive completes.

Unofficial Draft for Comment Only

40 41

42

44

47

₄₃ ticket238-J.

 $_{45}$ ticket 238-J.

46 ticket236-H.

 $_{48}$ ticket238-J.

"Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (End of advice to users.)

3.7.3 Communication Completion

The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communica-6 tion. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was 10 used, the completion of the send operation indicates that a matching receive was initiated, 11 and that the message will eventually be received by this matching receive. 12

The completion of a receive operation indicates that the receive buffer contains the 13 received message, the receiver is now free to access it, and that the status object is set. It 14does not indicate that the matching send operation has completed (but indicates, of course, 15that the send was initiated). 16

We shall use the following terminology: A **null** handle is a handle with value 17MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the re-18 quest is not associated with any ongoing communication (see Section 3.9). A handle is 19 active if it is neither null nor inactive. An empty status is a status which is set to re-20turn tag = MPI_ANY_TAG , source = MPI_ANY_SOURCE , error = $MPI_SUCCESS$, and is also 21ticket265. 22 internally configured so that calls to MPI_GET_COUNT[and], MPI_GET_ELEMENTS, and ticket265. 23 $MPI_GET_ELEMENTS_X$ return count = 0 and $MPI_TEST_CANCELLED$ returns false. We set a status variable to empty when the value returned by it is not significant. Status is set 24in this way so as to prevent errors due to accesses of stale information. 25

> The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

> Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_STATUS. For the functions MPI_TEST, MPI_TESTANY, MPI_WAIT, and MPI_WAITANY, which return a single MPI_STATUS value, the normal MPI error return process should be used (not the MPI_ERROR field in the MPI_STATUS argument).

MPI_WAIT(request, status)

```
request (handle)
                  INOUT
                            request
           39
           40
                   OUT
                                                       status object (Status)
                            status
           41
           42
                 int MPI_Wait(MPI_Request *request, MPI_Status *status)
ticket-248T. 43
                MPI_Wait(request, status, ierror) BIND(C)
           44
                     TYPE(MPI_Request), INTENT(INOUT) :: request
           45
                     TYPE(MPI_Status) :: status
           46
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           47
           48
```

1

 $\mathbf{2}$

3 4

5

7

8

9

26

27

28

29

30

 31

32

33

34

35 36 37

MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

{void MPI::Request::Wait(MPI::Status& status) (binding deprecated, see Section 15.2 }

{void MPI::Request::Wait()(binding deprecated, see Section 15.2) }

A call to MPI_WAIT returns when the operation identified by request is complete. If 8 the communication object associated with this request was created by a nonblocking send 9 or receive call, then the object request is an active persistent request, it is marked inactive. 10 Any other type of request is [deallocated by the call to MPI_WAIT] and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation. 12

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status.

Advice to users. Successful return of MPI_WAIT after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to free program space that was committed to the communication subsystem). (End of advice to users.)

Advice to implementors. In a multi-threaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (End of advice to implementors.)

MPI_TEST(request, flag, status)

INOUT	request	communication request (handle)
OUT	flag	true if operation completed (logical)
OUT	status	status object (Status)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) MPI_Test(request, flag, status, ierror) BIND(C) TYPE(MPI_Request), INTENT(INOUT) :: request

LOGICAL, INTENT(OUT) :: flag TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_TEST(REQUEST, FLAG, STATUS, IERROR)

LOGICAL FLAG INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Unofficial Draft for Comment Only

1

2

5 6

7

13

14

15

16

17

18 19

20

21

22

23

 24

25

2627

28

29

30 3132

39

42

43

44

4546

47

48

ticket321.

11 ticket321.

 $_{40}$ ticket 229.1. 41 ticket-248T.

1 2	<pre>{bool MPI::Request::Test(MPI::Status& status)(binding deprecated, see Section 15.2) }</pre>
3	<pre>{bool MPI::Request::Test()(binding deprecated, see Section 15.2) }</pre>
5 ticket321. 6 7 8 9	A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation[; if the communication object was created by a nonblocking send or receive, then it]. If the request is an active persistent request, it is marked as inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag
ticket321. $^{10}_{11}$	= false[, otherwise.] if the operation identified by request is not complete. In this case, the value of the status object is undefined. MPI_TEST is a local operation.
12 13	The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information
14 15	that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8). One is allowed to call MPI_TEST with a null or inactive request argument. In such a
16 17	case the operation returns with $flag = true$ and empty status. The functions MPI_WAIT and MPI_TEST can be used to complete both sends and
18 19	receives.
20 21 22	Advice to users. The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be empleted with periodic calls to MPI_TEST (End of advice to
22 23 24	thread scheduler can be emulated with periodic calls to MPI_TEST. (<i>End of advice to users.</i>)
25 26	Example 3.12 Simple usage of nonblocking operations and MPI_WAIT.
27 28	CALL MPI_COMM_RANK(comm, rank, ierr) IF (rank.EQ.0) THEN
29 30	CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr) **** do some computation to mask latency ****
31 32 33	CALL MPI_WAIT(request, status, ierr) ELSE IF (rank.EQ.1) THEN
34 35	CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr) **** do some computation to mask latency ****
36 37	CALL MPI_WAIT(request, status, ierr) END IF
38 39	A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.
40 41	
42 43	MPI_REQUEST_FREE(request) INOUT request communication request (handle)
44 45	<pre>int MPI_Request_free(MPI_Request *request)</pre>
ticket-248T. 46 47 48	MPI_Request_free(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(INOUT) :: request

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_REQUEST_FREE(REQUEST, IERROR)	$\frac{2}{3}$
INTEGER REQUEST, IERROR	4
<pre>{void MPI::Request::Free()(binding deprecated, see Section 15.2) }</pre>	5
Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An	6
ongoing communication that is associated with the request will be allowed to complete. The	7
request will be deallocated only after its completion.	8 9
	10
Rationale. The MPI_REQUEST_FREE mechanism is provided for reasons of perfor-	11
mance and convenience on the sending side. (End of rationale.)	12
Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not	13
possible to check for the successful completion of the associated communication with	14
calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the	15
communication, an error code cannot be returned to the user — such an error must	16 17
be treated as fatal. An active receive request should never be freed as the receiver	18
will have no way to verify that the receive has completed and the receive buffer can	19
be reused. (End of advice to users.)	20
	21
Example 3.13 An example using MPI_REQUEST_FREE.	22
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)	23 24
IF (rank.EQ.0) THEN	25
DO i=1, n	26
CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	27
CALL MPI_REQUEST_FREE(req, ierr)	28
CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	29
CALL MPI_WAIT(req, status, ierr)	30
END DO	31 32
ELSE IF (rank.EQ.1) THEN CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	33
CALL MPI_WAIT(req, status, ierr)	34
DO I=1, n-1	35
CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	36
CALL MPI_REQUEST_FREE(req, ierr)	37
CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	38
CALL MPI_WAIT(req, status, ierr) END DO	39 40
END DU CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	40
CALL MPI_WAIT(req, status, ierr)	42
END IF	43
	44
3.7.4 Semantics of Nonblocking Communications	45
The semantics of nonblocking communication is defined by suitably extending the definitions in Section 3.5 .	46 47 48

1 **Order** Nonblocking communication operations are ordered according to the execution order $\mathbf{2}$ of the calls that initiate the communication. The non-overtaking requirement of Section 3.53 is extended to nonblocking communication, with this definition of order being used. 4 Example 3.14 Message ordering for nonblocking operations. 56 CALL MPI_COMM_RANK(comm, rank, ierr) 7 IF (RANK.EQ.0) THEN 8 CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr) 9 CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr) 10 ELSE IF (rank.EQ.1) THEN 11 CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr) 12CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr) 13 END IF 14CALL MPI_WAIT(r1, status, ierr) 15CALL MPI_WAIT(r2, status, ierr) 1617The first send of process zero will match the first receive of process one, even if both messages 18 are sent before process one executes either receive. 1920**Progress** A call to MPI_WAIT that completes a receive will eventually terminate and return 21if a matching send has been started, unless the send is satisfied by another receive. In 22particular, if the matching send is nonblocking, then the receive should complete even if no 23call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that 24 completes a send will eventually return if a matching receive has been started, unless the 25receive is satisfied by another send, and even if no call is executed to complete the receive. 2627Example 3.15 An illustration of progress semantics. 28CALL MPI_COMM_RANK(comm, rank, ierr) 29IF (RANK.EQ.O) THEN 30 CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr) 31CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr) 32 ELSE IF (rank.EQ.1) THEN 33 CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr) 34 CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr) 35 CALL MPI_WAIT(r, status, ierr) 36 END IF 37 38 This code should not deadlock in a correct MPI implementation. The first synchronous 39 send of process zero must complete after process one posts the matching (nonblocking) 40receive even if process one has not yet reached the completing wait call. Thus, process zero 41 will continue and execute the second send, allowing process one to complete execution.

⁴² If an MPI_TEST that completes a receive is repeatedly called with the same arguments, ⁴³ and a matching send has been started, then the call will eventually return flag = true, unless ⁴⁴ the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly ⁴⁵ called with the same arguments, and a matching receive has been started, then the call will ⁴⁶ eventually return flag = true, unless the receive is satisfied by another send.

3.7.5 Mu	1		
It is conve	the completion of any, some, or all the operations	2	
		r a specific message. A call to MPI_WAITANY or	3
	=	the completion of one out of several operations. A	4
		can be used to wait for all pending operations in	5
		PI_TESTSOME can be used to complete all enabled	6
operations in a list.			7
- F			8
			9
MPI_WAIT	ANY (count, array_of_request	s, index, status)	10
IN	count	list length (non-negative integer)	11
INOUT	arrest of requests		12 13
	array_of_requests	array of requests (array of handles)	13
OUT	index	index of handle for operation that completed (integer)	14
OUT	status	status object (Status)	16
			17
int MPI_W	aitany(int count, MPI_Re	<pre>quest [*]array_of_requests[], int *index,</pre>	18 ticket125.
	MPI_Status *status)		$_{19}$ ticket 125.
MPI_Waita	$_{20}$ ticket-248T.		
INTEG	21		
TYPE(22		
INTEG	23		
TYPE(24		
	25		
	ER, OPTIONAL, INTENT(OUT		26
		STS, INDEX, STATUS, IERROR)	27
		STS(*), INDEX, STATUS(MPI_STATUS_SIZE),	28
IERRC	IR		29
{static i	nt MPI::Request::Waitany	(int count,	30
ι.		f_requests[], MPI::Status& status)(binding	31
	deprecated, see Section	•	32
(33
{static i	nt MPI::Request::Waitany		34
	<pre>MP1::Request array_o Section 15.2) }</pre>	f_requests[])(binding deprecated, see	35
	36		
Blocks	37		

Blocks until one of the operations associated with the active requests in the array has completed. If more then one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing [communication]operation. (The array is indexed from zero in C, and from one in Fortran.) If the request [was allocated by a nonblocking communication operation, then it] is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL.

The array_of_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI_UNDEFINED, and a empty status.

38

39

40

41

43

44

45

46

 $47 \\ 48$

ticket321.

 $_{42}$ ticket 321.

```
1
                      The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same
            \mathbf{2}
                 effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value
            3
                 returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an
            4
                 array containing one active entry is equivalent to MPI_WAIT.
            5
            6
                 MPI_TESTANY(count, array_of_requests, index, flag, status)
            \overline{7}
            8
                                                          list length (non-negative integer)
                   IN
                             count
            9
                   INOUT
                             array_of_requests
                                                          array of requests (array of handles)
            10
                   OUT
                             index
                                                          index of operation that completed, or
            11
                                                          MPI_UNDEFINED if none completed (integer)
           12
                   OUT
                             flag
                                                          true if one of the operations is complete (logical)
           13
            14
                   OUT
                             status
                                                          status object (Status)
            15
            16
  ticket125.
                 int MPI_Testany(int count, MPI_Request [*]array_of_requests[], int *index,
            17
  ticket125.
                                 int *flag, MPI_Status *status)
           18
ticket-248T.
            19
                 MPI_Testany(count, array_of_requests, index, flag, status, ierror) BIND(C)
                      INTEGER, INTENT(IN) :: count
           20
                      TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
           21
                      INTEGER, INTENT(OUT) :: index
           22
           23
                      LOGICAL, INTENT(OUT) :: flag
            24
                      TYPE(MPI_Status) :: status
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            25
           26
                 MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
           27
                      LOGICAL FLAG
           28
                      INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
           29
                      IERROR
           30
                 {static bool MPI::Request::Testany(int count,
           31
           32
                                 MPI::Request array_of_requests[], int& index,
           33
                                 MPI::Status& status) (binding deprecated, see Section 15.2) }
           34
                 {static bool MPI::Request::Testany(int count,
           35
                                 MPI::Request array_of_requests[], int& index) (binding deprecated,
           36
                                 see Section 15.2 }
           37
           38
                      Tests for completion of either one or none of the operations associated with active
           39
                 handles. In the former case, it returns flag = true, returns in index the index of this
  ticket321. 40
                 request in the array, and returns in status the status of that operation; if the request was
           41
                 allocated by a nonblocking communication call then the request. If the request is an active
           42
                 persistent request, it is marked as inactive. Any other type of request is deallocated and
           43
                 the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from
           ^{44}
                 one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns
           45
                 a value of MPI_UNDEFINED in index and status is undefined.
           46
                      The array may contain null or inactive handles. If the array contains no active handles
           47
                 then the call returns immediately with flag = true, index = MPI_UNDEFINED, and an empty
           48
                 status.
```

3.7. NONBLOCKING COMMUNICATION

1 If the array of requests contains active handles then the execution of $\mathbf{2}$ MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_TEST(&array_of_requests[i], flag, status), for i=0, 1, ..., count-1, in some arbitrary 3 4 order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an 56 array containing one active entry is equivalent to MPI_TEST. 7 8 MPI_WAITALL(count, array_of_requests, array_of_statuses) 9 10 IN count lists length (non-negative integer) 11 INOUT array_of_requests array of requests (array of handles) 12OUT array_of_statuses array of status objects (array of Status) 13 14int MPI_Waitall(int count, MPI_Request [*]array_of_requests[], ¹⁵ ticket125. MPI_Status [*]array_of_statuses[]) ¹⁶ ticket125. 17 ticket125. MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) BIND(C) $_{18}$ ticket 125. INTEGER, INTENT(IN) :: count 19 ticket-248T. TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) 20TYPE(MPI_Status) :: array_of_statuses(*) 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) 23INTEGER COUNT, ARRAY_OF_REQUESTS(*) 24 25INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 26{static void MPI::Request::Waitall(int count, 27MPI::Request array_of_requests[], 28MPI::Status array_of_statuses[]) (binding deprecated, see 29Section 15.2 } 30 31{static void MPI::Request::Waitall(int count, 32 MPI::Request array_of_requests[]) (binding deprecated, see 33 Section 15.2 } 34 Blocks until all communication operations associated with active handles in the list 35

blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The ith entry in array_of_statuses is set to the return status of the i-th operation. [Requests that were created by nonblocking communication operations]Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI_WAITALL(count, array_of_requests, array_of_statuses) has the same effect as the execution of MPI_WAIT(&array_of_request[i], &array_of_statuses[i]), for i=0 ,..., count-1, in some arbi-

trary order. MPI_WAITALL with an array of length one is equivalent to MPI_WAIT.

When one or more of the communications completed by a call to MPI_WAITALL fail, it is desireable to return specific information on each communication. The function

36

37

39

40

41

42

43

44

45

46

47

48

38 ticket321.

1 2 3 4 5 6 7 8 9 10 11 11 12 13	 error field of each status to a specific error code. This code will be MPI_SUCCESS, if specific communication completed; it will be another specific error code, if it failed; or it of be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITA will return MPI_SUCCESS if no request had an error, or will return another error code i failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses. <i>Rationale.</i> This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. needs to check each individual status only when an error occurred. (<i>End of rational</i> 23) 				
14 15	MPI_TEST	TALL(count, array_of_req	uests, flag, array_of_statuses)		
16	IN	count	lists length (non-negative integer)		
17	INOUT	array_of_requests	array of requests (array of handles)		
18 19	OUT	flag	(logical)		
20	OUT	array_of_statuses	array of status objects (array of Status)		
21					
ticket125. ²² ticket125. ²³ ticket125. ²⁴	int MPI_7		<pre>PI_Request [*]array_of_requests[], int *flag, rray_of_statuses[])</pre>		
ticket-248T. 26		BIND(C)	requests, flag, array_of_statuses, ierror)		
27		ER, INTENT(IN) :: c MPI Request) INTENT	count C(INOUT) :: array_of_requests(count)		
28 29		CAL, INTENT(OUT) ::	· ·		
29 30			_Status) :: array_of_statuses(*)		
31	INTEG	ER, OPTIONAL, INTENT	'(OUT) :: ierror		
32	MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)				
33 34		LOGICAL FLAG INTEGER COUNT, ARRAY_OF_REQUESTS(*),			
35		LER COUNT, ARRAY_OF_R [OF_STATUSES(MPI_STA			
36		oool MPI::Request::Te			
37 38	istatic t	-	ray_of_requests[],		
39		-	ay_of_statuses[]) (binding deprecated, see		
40		Section 15.2 }			
41	$\{ \texttt{static} \ \texttt{t} \}$	oool MPI::Request::Te	stall(int count,		
42 43		-	ray_of_requests[])(binding deprecated, see		
44		Section 15.2 }			
45			munications associated with active handles in the array		
$^{46}_{ m ticket 321.~_{47}}$	-		case where no handle in the list is active). In this case, s to an active [handle]request is set to the status of the		
ticket $321{47}$ ticket $321{48}$			he request was allocated by a nonblocking communication		

call then it is deallocated, and the handle is]operation. Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty. Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a local operation. Errors that occurred during the execution of MPI_TESTALL are handled as errors in MPI_WAITALL.					
MPI_WAI	TSOME(incount, array_of_	requests, outcount, array_of_indices, array_of_statuses)	10 11 12		
IN	incount	length of array_of_requests (non-negative integer)	12		
INOUT	array_of_requests	array of requests (array of handles)	14		
		· - · · · /	15		
OUT	outcount	number of completed requests (integer)	16		
OUT	array_of_indices	array of indices of operations that completed (array of integers)	17 18		
OUT	array_of_statuses	array of status objects for operations that completed	19		
001		(array of Status)	20		
		(drive or status)	21		
int MPT	Waitsome(int incount.	<pre>MPI_Request [*]array_of_requests[],</pre>	$^{22}_{_{23}}$ ticket125.		
		<pre>nt [*]array_of_indices[],</pre>	$_{24}^{23}$ ticket 125.		
		ray_of_statuses[])	$^{24}_{25}$ ticket 125.		
	<pre>some(incount, array_of</pre>	_requests, outcount, array_of_indices, s, ierror) BIND(C)	26 ticket125. 26 ticket125. 27 ticket125.		
	GER, INTENT(IN) :: in	count INOUT) :: array_of_requests(incount)	²⁸ ticket-248T ²⁹		
	-	utcount, array_of_indices(*)	30		
	(MPI_Status) :: array	•	31		
	GER, OPTIONAL, INTENT(32		
			33		
MPI_WAIT		_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,	34		
T. 1100	ARRAY_OF_STATUSES	-	35		
		REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	36		
AKKA	Y_OF_STATUSES(MPI_STAT	US_SIZE,*), IERRUR	37		
{static	int MPI::Request::Wait	some(int incount,	38		
	MPI::Request arra	<pre>ay_of_requests[], int array_of_indices[],</pre>	39		
	•	<pre>y_of_statuses[])(binding deprecated, see</pre>	40		
	Section 15.2 }		41		
{static	int MPI::Request::Wait	some(int incount.	42 43		
[200020	MPI::Request arra		43		
	-	Les []) (binding deprecated, see Section 15.2) }	45		
TT7 •	·		46		
	Waits until at least one of the operations associated with active handles in the list have				
completed. Returns in outcount the number of requests from the list array_of_requests that					

Unofficial Draft for Comment Only

1 have completed. Returns in the first outcount locations of the array array_of_indices the $\mathbf{2}$ indices of these operations (index within the array array_of_requests; the array is indexed 3 from zero in C and from one in Fortran). Returns in the first outcount locations of the array ticket321. 4 array_of_status the status for these completed operations. [If a request that completed was $\mathbf{5}$ allocated by a nonblocking communication call, then it Completed active persistent requests 6 are marked as inactive. Any other type or request that completed is deallocated, and the $\overline{7}$ associated handle is set to MPI_REQUEST_NULL. 8 If the list contains no active handles, then the call returns immediately with outcount 9 = MPI_UNDEFINED. 10 When one or more of the communications completed by MPI_WAITSOME fails, then 11it is desirable to return specific information on each communication. The arguments 12outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of 13all communications that have succeeded or failed. The call will return the error code 14MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate 15success or to indicate the specific error that occurred. The call will return MPI_SUCCESS 16if no request resulted in an error, and will return another error code if it failed for other 17reasons (such as invalid arguments). In such cases, it will not update the error fields of the 18 statuses. 1920MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses) 212223IN incount length of array_of_requests (non-negative integer) 24INOUT array_of_requests array of requests (array of handles) 25OUT outcount number of completed requests (integer) 2627OUT array_of_indices array of indices of operations that completed (array of 28integers) 29OUT array_of_statuses array of status objects for operations that completed 30 (array of Status) 31 32 ticket125. 33 int MPI_Testsome(int incount, MPI_Request [*]array_of_requests[], ticket 125. $_{34}$ int *outcount, int [*]array_of_indices[], ticket125. 35 MPI_Status [*]array_of_statuses[]) ticket125. 36 MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, ticket125. 37array_of_statuses, ierror) BIND(C) ticket125. ticket-248T. 38 INTEGER, INTENT(IN) :: incount 39 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount) 40INTEGER, INTENT(OUT) :: outcount, array_of_indices(*) 41 TYPE(MPI_Status) :: array_of_statuses(*) 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 44 ARRAY_OF_STATUSES, IERROR) 45INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 46 ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 4748

<pre>{static int MPI::Request::Testsome(int incount,</pre>	1
<pre>MPI::Request array_of_requests[], int array_of_indices[],</pre>	2
<pre>MPI::Status array_of_statuses[])(binding deprecated, see</pre>	3
Section 15.2 }	4
<pre>{static int MPI::Request::Testsome(int incount,</pre>	5
MPI::Request array_of_requests[],	6
int array_of_indices[]) (binding deprecated, see Section 15.2) }	7
int array_or_indices[]) (binding deprecated, see Section 15.2) }	8
Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has	9
completed it returns $outcount = 0$. If there is no active handle in the list it returns $outcount$	10
= MPI_UNDEFINED.	11
MPI_TESTSOME is a local operation, which returns immediately, whereas	12
MPI_WAITSOME will block until a communication completes, if it was passed a list that	13
contains at least one active handle. Both calls fulfill a fairness requirement: If a request for	14
a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or	15
MPI_TESTSOME, and a matching send has been posted, then the receive will eventually	16
succeed, unless the send is satisfied by another receive; and similarly for send requests.	17
Errors that occur during the execution of MPI_TESTSOME are handled as for	18
MPI_WAITSOME.	19
	20
Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use	21
of MPI_TESTANY. The former returns information on all completed communications,	22
with the latter, a new call is required for each communication that completes.	23
A server with multiple clients can use MPI_WAITSOME so as not to starve any client.	24
Clients send messages to the server with service requests. The server calls	25
MPI_WAITSOME with one receive request for each client, and then handles all receives	26
that completed. If a call to MPI_WAITANY is used instead, then one client could starve	27
while requests from another client always sneak in first. (End of advice to users.)	28
	29
Advice to implementors. MPI_TESTSOME should complete as many pending com-	30
munications as possible. (End of advice to implementors.)	31
	32
Example 3.16 Client-server code (starvation can occur).	33
Example 5.16 Chent server code (starvation can occur).	34
	35
CALL MPI_COMM_SIZE(comm, size, ierr)	36
CALL MPI_COMM_RANK(comm, rank, ierr)	37
IF(rank .GT. 0) THEN ! client code	38
DO WHILE(.TRUE.)	39
CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)	40
CALL MPI_WAIT(request, status, ierr)	41
END DO	42
ELSE ! rank=0 server code	43
DO i=1, size-1	44
CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,	45
<pre>comm, request_list(i), ierr)</pre>	46
END DO	47
DO WHILE(.TRUE.)	48

```
1
                 CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
\mathbf{2}
                 CALL DO_SERVICE(a(1,index)) ! handle one message
3
                 CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
4
                            comm, request_list(index), ierr)
5
             END DO
6
     END IF
7
8
                       Same code, using MPI_WAITSOME.
     Example 3.17
9
10
11
     CALL MPI_COMM_SIZE(comm, size, ierr)
12
     CALL MPI_COMM_RANK(comm, rank, ierr)
13
     IF(rank .GT. 0) THEN
                                      ! client code
14
          DO WHILE(.TRUE.)
15
             CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
16
             CALL MPI_WAIT(request, status, ierr)
17
          END DO
18
     ELSE
                    ! rank=0 -- server code
19
          DO i=1, size-1
20
             CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
21
                              comm, request_list(i), ierr)
22
          END DO
23
          DO WHILE(.TRUE.)
24
             CALL MPI_WAITSOME(size, request_list, numdone,
25
                                indices, statuses, ierr)
26
             DO i=1, numdone
27
                 CALL DO_SERVICE(a(1, indices(i)))
28
                 CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
29
                               comm, request_list(indices(i)), ierr)
30
             END DO
31
          END DO
32
     END IF
33
34
            Non-destructive Test of status
     3.7.6
35
     This call is useful for accessing the information associated with a request, without freeing
36
     the request (in case the user is expected to access it later). It allows one to layer libraries
37
     more conveniently, since multiple layers of software may access the same completed request
38
     and extract from it the status information.
39
40
41
     MPI_REQUEST_GET_STATUS( request, flag, status )
42
       IN
                                             request (handle)
                 request
43
44
       OUT
                                             boolean flag, same as from MPI_TEST (logical)
                 flag
45
       OUT
                 status
                                             MPI_STATUS object if flag is true (Status)
46
47
48
```

```
1
int MPI_Request_get_status(MPI_Request request, int *flag,
                                                                                     2
              MPI_Status *status)
                                                                                      ticket-248T.
                                                                                     3
MPI_Request_get_status(request, flag, status, ierror) BIND(C)
                                                                                     4
    TYPE(MPI_Request), INTENT(IN) :: request
                                                                                     5
    LOGICAL, INTENT(OUT) :: flag
                                                                                     6
    TYPE(MPI_Status) :: status
                                                                                     7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     9
MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
                                                                                     10
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                     11
    LOGICAL FLAG
                                                                                     12
{bool MPI::Request::Get_status(MPI::Status& status) const(binding deprecated,
                                                                                     13
              see Section 15.2 }
                                                                                     14
                                                                                     15
{bool MPI::Request::Get_status() const(binding deprecated, see Section 15.2) }
                                                                                     16
```

Sets flag=true if the operation is complete, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or inactivate the request; a subsequent call to test, wait or free should be executed with that request. It sets flag=false if the operation is not complete.

One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request argument. In such a case the operation returns with flag=true and empty status.

Probe and Cancel 3.8

The MPI_PROBE and , MPI_IPROBE, MPI_MPROBE, and MPI_IMPROBE operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

The MPI_CANCEL operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

3.8.1 Probe

MPI_IPROBE(source, tag, comm, flag, status)				
IN	source	rank of source or MPI_ANY_SOURCE (integer)	39	
IN	tag		40	
	tag	message tag or MPI_ANY_TAG (integer)	41	
IN	comm	communicator (handle)	42	
OUT	flag	(logical)	43	
OUT	status	status object (Status)	44	
001	Status	status object (status)	45 46	
int NDT Transla (int courses int to a NDT Commentation int offer				
<pre>int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,</pre>				

MPI_Status *status)

17

18

19

20

21

22 2324

25

29 30

 31

32

34 35

36 37

 26 ticket 38.

²⁷ ticket38. ²⁸ ticket38.

 33 ticket 38.

	1				
tticket248.P.	2	MPI_Iprobe	e(source, tag, comm, flag	, status, ierror) BIND(C)	
	3	INTEGER, INTENT(IN) :: source, tag			
	4 TYPE(MPI_Comm), INTENT(IN) :: comm				
5 LOGICAL, INTENT(OUT) :: flag					
	6				
	7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
	8	MPI_IPROB	OBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)		
	9	, I.OGTCAL FLAG			
	¹⁰ INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IEI			TUS(MPI_STATUS_SIZE), IERROR	
	11		··Comm··Iprobe(int source	int tag MDIStatuck status)	
13		<pre>{bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status)</pre>			
	14				
	15	{bool MPI::Comm::Iprobe(int source, int tag) const(binding deprecated, see			
16		Section 15.2 }			
¹⁷ MPI_IPROB			PROBE(source, tag, comm, flag	g, status) returns $flag = true$ if there is a message	
	18	 that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(, source, tag, comm, status) executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined. If MPI_IPROBE returns flag = true, then the content of the status object can be sub- sequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message. A subsequent receive executed with the same communicator, and the source and tag 			
	19				
	20				
	21				
	26				
	27				
	28	if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multi-threaded, it is the user's			
	29				
	30				
	³¹ The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag a				
	³² can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and			robe for messages from an arbitrary source and/or	
	33	 with an arbitrary tag. However, a specific communication context must be provided with the comm argument. It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received. 			
	34				
	35 36				
	37				
38					
	³⁹ MPI_PROBE(source, tag, comm, status)				
	40	IN	source	rank of source or MPI_ANY_SOURCE (integer)	
	41	IN		message tag or MPI_ANY_TAG (integer)	
	42		tag		
	43	IN	comm	communicator (handle)	
	44	OUT	status	status object (Status)	
	45				
	46	<pre>int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)</pre>			
	cicket229.2.47 icket-248T.48 MPI_Probe(source, tag, comm, status, ierror) BIND(C)				
ticket-248T. ⁴⁸ MPI_Probe(source, tag, comm, status, ierror) BIND(C)			5, IGIIOI / DIND(0)		

INTEGER, INTENT(IN) :: source, tag	1							
TYPE(MPI_Comm), INTENT(IN) :: comm								
TYPE(MPI_Status) :: status	3							
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4							
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)								
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR								
<pre>{void MPI::Comm::Probe(int source, int tag, MPI::Status& status)</pre>								
{void MPI::Comm::Probe(int source, int tag) const(binding deprecated, see								
Section 15.2 }								
MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns								
only after a matching message has been found. The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress:								
				if a call to MPI_PROBE has been issued by a process, and a send that matches the probe				
has been initiated by some process, then the call to MPI_PROBE will return, unless the								
message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a matching message has been issued, then the call to MPI_IPROBE will eventually return flag								
					= true unless the message is received by another concurrent receive operation or matched			
					by a concurrent matched probe.			
Example 3.18								

```
Use blocking probe to wait for an incoming message.
```

CALL MPI_COMM_RANK(comm, rank, ierr) 27IF (rank.EQ.0) THEN 28CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr) 29ELSE IF (rank.EQ.1) THEN 30 CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr) 31 ELSE IF (rank.EQ.2) THEN 32 DO i=1, 2 33 CALL MPI_PROBE(MPI_ANY_SOURCE, 0, 34comm, status, ierr) 35 IF (status(MPI_SOURCE) .EQ. 0) THEN 36 100 CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr) 37 ELSE 38200 CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr) 39 END IF 40END DO 41 END IF 42

Each message is received with the right type.

Example 3.19 A similar program to the previous example, but now it has a problem.

47 48

43

 $44 \\ 45$

46

1	CALL MPI_COMM_RANK(comm, rank, ierr)				
2	IF (rank.EQ.0) THEN				
3	CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)				
4	ELSE IF (rank.EQ.1) THEN				
5	•				
	CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)				
6	ELSE IF (rank.EQ.2) THEN				
7	DO i=1, 2				
8	CALL MPI_PROBE(MPI_ANY_SOURCE, 0,				
9	comm, status, ierr)				
10	IF (status(MPI_SOURCE) .EQ. 0) THEN				
11	100 CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,				
12					
13	0, comm, status, ierr)				
	ELSE				
14	200 CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,				
15	0, comm, status, ierr)				
16	END IF				
17	END DO				
18	END IF				
ticket262. $_{19}$					
20	[We slightly modified Example 3.18, using MPI_ANY_SOURCE as the source argument				
20	in the two receive calls in statements labeled 100 and 200. The program is now incorrect:				
	the receive operation may receive a message that is distinct from the message probed by				
22					
23	the preceding call to MPI_PROBE.]In Example 3.19, the two receive calls in statements				
24	labeled 100 and 200 in Example 3.18 slightly modified, using MPI_ANY_SOURCE as the source				
25	argument. The program is now incorrect: the receive operation may receive a message that				
ticket 38. $_{26}$	is distinct from the message probed by the preceding call to MPI_PROBE.				
27	Advice to users In a multithreaded MDI program MDI DDODE and				
28	Advice to users. In a multithreaded MPI program, MPI_PROBE and				
29	MPI_IPROBE might need special care. If a thread probes for a message and then				
	immediately posts a matching receive, the receive may match a message other than				
30	that found by the probe since another thread could concurrently receive that original				
31	message [29]. MPI_MPROBE and MPI_IMPROBE solve this problem by matching the				
32	incoming message so that it may only be received with MPI_MRECV or MPI_IMRECV				
33	on the corresponding message handle. (End of advice to users.)				
34					
35	Advise to implementance A call to MDL DDODE (course to grown status) will match				
36	Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match				
37	the message that would have been received by a call to MPI_RECV(, source, tag,				
	comm, status) executed at the same point. Suppose that this message has source s,				
38	tag t and communicator c . If the tag argument in the probe call has value				
39	MPI_ANY_TAG then the message probed will be the earliest pending message from				
40	source s with communicator c and any tag; in any case, the message probed will be				
41	the earliest pending message from source s with tag t and communicator c (this is the				
42					
43	message that would have been received, so as to preserve message order). This message				
44	continues as the earliest pending message from source s with tag t and communicator				
45	c, until it is received. A receive operation subsequent to the probe that uses the				
	same communicator as the probe and uses the tag and source values returned by				
46	the probe, must receive this message, unless it has already been received by another				
47	receive operation. (End of advice to implementors.)				
ticket38 ⁴⁸					

ticket 38. $^{\scriptscriptstyle 48}$

3.8.2 Matching Probe

The function MPI_PROBE checks for incoming messages without receiving them. Since the list of incoming messages is global among the threads of each MPI process, it can be hard to use this functionality in threaded environments [29, 26].

Like MPI_PROBE and MPI_IPROBE, the MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message that was matched regardless of other intervening probe or receive operations. This gives the application an opportunity to decide how to receive the message, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

MPI_IMPROBE(source, tag, comm, flag, message, status)

IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	flag	flag (logical)
OUT	message	returned message (handle)
OUT	status	status object (Status)

MPI_Improbe(source, tag, comm, flag, message, status, ierror) BIND(C)
INTEGER, INTENT(IN) :: source, tag
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: flag
TYPE(MPI_Message), INTENT(OUT) :: message
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS(MPI_STATUS_SIZE),
IERROR

MPI_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the program and returns in status the same value that would have been returned by MPI_RECV. In addition, it returns in message a handle to the matched message. Otherwise, the call returns flag = false, and leaves status and message undefined.

A matched receive (MPI_MRECV or MPI_IMRECV) executed with the message handle will receive the message that was matched by the probe. Unlike MPI_IPROBE, no other probe or receive operation may match the message returned by MPI_IMPROBE. ⁴⁷

 $\mathbf{2}$

 26 ticket-248T.

```
1
                 Each message returned by MPI_IMPROBE must be received with either MPI_MRECV or
            \mathbf{2}
                 MPI_IMRECV.
            3
                     The source argument of MPI_IMPROBE can be MPI_ANY_SOURCE, and the tag argu-
            4
                 ment can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source
            \mathbf{5}
                 and/or with an arbitrary tag. However, a specific communication context must be provided
            6
                 with the comm argument.
            7
                     A synchronous send operation that is matched with MPI_IMPROBE or MPI_MPROBE
            8
                 will complete successfully only if both a matching receive is posted with MPI_MRECV or
            9
                 MPI_IMRECV, and the receive operation has started to receive the message sent by the
           10
                 synchronous send.
           11
                     A matching probe with MPI_PROC_NULL as source returns flag = true.
           12
                 message = MPI_MESSAGE_NULL, and the status object returns source = MPI_PROC_NULL,
                 tag = MPI_ANY_TAG, and count = 0; see Section 3.11.
           13
           14
           15
                 MPI_MPROBE(source, tag, comm, message, status)
           16
           17
                   IN
                                                        rank of source or MPI_ANY_SOURCE (integer)
                            source
           18
                   IN
                                                        message tag or MPI_ANY_TAG (integer)
                            tag
           19
                   IN
                            comm
                                                        communicator (handle)
           20
           21
                   OUT
                                                        returned message (handle)
                            message
           22
                   OUT
                            status
                                                        status object (Status)
           23
           ^{24}
                 int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
           25
                                MPI_Status *status)
           26
ticket-248T.
           27
                 MPI_Mprobe(source, tag, comm, message, status, ierror) BIND(C)
           28
                     INTEGER, INTENT(IN) :: source, tag
           29
                     TYPE(MPI_Comm), INTENT(IN) :: comm
           30
                     TYPE(MPI_Message), INTENT(OUT) :: message
           ^{31}
                     TYPE(MPI_Status) :: status
           32
                     INTEGER, OPTIONAL, INTENT(OUT) ::
                                                            ierror
           33
                 MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
           34
                     INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
           35
           36
                     MPI_MPROBE behaves like MPI_IMPROBE except that it is a blocking call that returns
           37
                 only after a matching message has been found.
           38
                     The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progress
           39
                 in the same way as in the case of MPI_PROBE and MPI_IPROBE.
           40
           41
                        Matched Receives
                 3.8.3
           42
                 The functions MPI_MRECV and MPI_IMRECV receive messages that have been previously
           43
                 matched by a matching probe (Section 3.8.2).
           44
           45
           46
           47
           48
```

MPI_MRE	CV(buf, count, datatype, mess	age, status)	1
OUT	buf	initial address of receive buffer (choice)	2
IN	count	number of elements in receive buffer (non-negative in-	3
IIN	count	teger)	4
INI	detet we		5
IN	datatype	datatype of each receive buffer element (handle)	7
INOUT	message	message (handle)	8
OUT	status	status object (Status)	9
			10
int MPI_N	Mrecv(void* buf, int coun	t, MPI_Datatype datatype,	11
	MPI_Message *message	e, MPI_Status *status)	12
MPT Mrecy	v(buf, count, datatype, m	essage, status, ierror) BIND(C)	$_{13}$ ticket-248T.
	(*), DIMENSION() :: b	· · · · · · · · · · · · · · · · · · ·	14
	GER, INTENT(IN) :: count		15
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	16 17
	(MPI_Message), INTENT(INO	UT) :: message	18
	(MPI_Status) :: status		19
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_MRECV	V(BUF, COUNT, DATATYPE, M	ESSAGE, STATUS, IERROR)	21
<type< td=""><td>e> BUF(*)</td><td></td><td>22</td></type<>	e> BUF(*)		22
INTE	GER COUNT, DATATYPE, MESS	AGE, STATUS(MPI_STATUS_SIZE), IERROR	23
This	call receives a message match	ed by a matching probe operation (Section $3.8.2$).	24
	0	torage containing count consecutive elements of the	25
		ldress buf. The length of the received message must	26 27
		the receive buffer. An overflow error occurs if all	27
		ncation, into the receive buffer.	29
	0	ceive buffer, then only those locations corresponding	30
	orter) message are modified.		31
		message handle is set to MPI_MESSAGE_NULL. All	32
	0	of this operation are handled according to the error in the matching probe call that produced the message	33
handle.	t for the communicator used in	the matching probe can that produced the message	34
	MRECV is called with MPI	MESSAGE_NULL as the message argument, the call	35
		ject set to source = MPI_PROC_NULL , tag =	36 37
MPI_ANY_	TAG, and count $= 0$, as if a	receive from MPI_PROC_NULL was issued, see Sec-	38
tion 3.11 .			39
			40
			41
			42
			43
			44
			45
			46
			47 48
			40

```
1
                 MPI_IMRECV(buf, count, datatype, message, request)
            2
                   OUT
                             buf
                                                        initial address of receive buffer (choice)
            3
                   IN
                            count
                                                        number of elements in receive buffer (non-negative in-
            4
                                                         teger)
            5
            6
                   IN
                            datatype
                                                        datatype of each receive buffer element (handle)
            7
                   INOUT
                                                        message (handle)
                             message
            8
                   OUT
                             request
                                                        communication request (handle)
            9
            10
                 int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype,
           11
                                MPI_Message *message, MPI_Request *request)
            12
ticket-248T. 13
                 MPI_Imrecv(buf, count, datatype, message, request, ierror) BIND(C)
            14
                     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
           15
                     INTEGER, INTENT(IN) :: count
           16
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
            17
                     TYPE(MPI_Message), INTENT(INOUT) :: message
            18
                     TYPE(MPI_Request), INTENT(OUT) :: request
            19
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           20
                 MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
           21
           22
                      <type> BUF(*)
                     INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR
           23
           24
                     MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking
           25
                 receive of a matched message. Completion semantics are similar to MPI_IRECV as described
            26
                 in Section 3.7.2. On return from this function, the message handle is set to
           27
                 MPI_MESSAGE_NULL.
           28
           29
                      Advice to implementors. If reception of a matched message is started with
           30
                      MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If
           31
                      MPI_CANCEL succeeds, the matched message must be found by a subsequent message
           32
                      probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by
           33
                      a subsequent receive operation or canceled by the sender. See Section 3.8.4 for details
           34
                      about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may
           35
                      fail. (End of advice to implementors.)
           36
           37
                 3.8.4 Cancel
           38
           39
            40
                 MPI_CANCEL(request)
           41
                   IN
                             request
                                                        communication request (handle)
           42
           43
                 int MPI_Cancel(MPI_Request *request)
            44
ticket-248T. _{45}
                 MPI_Cancel(request, ierror) BIND(C)
            46
                     TYPE(MPI_Request), INTENT(IN) :: request
           47
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           48
```

MPI_CANCEL(REQUEST, IERROR) INTEGER REQUEST, IERROR

{void MPI::Request::Cancel() const(binding deprecated, see Section 15.2) }

A call to MPI_CANCEL marks for cancellation a pending, nonblocking communication operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled. It is still necessary to complete a communication that has been marked for cancellation, using a call to MPI_REQUEST_FREE, MPI_WAIT or MPI_TEST (or any of the derived operations).

If a communication is marked for cancellation, then a MPI_WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly called in a busy wait loop for a canceled communication, then MPI_TEST will eventually be successful.

MPI_CANCEL can be used to cancel a communication that uses a persistent request (see Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation cancels the active communication, but not the request itself. After the call to MPI_CANCEL and the subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message.

Either the cancellation succeeds, or the communication succeeds, but not both. If a send is marked for cancellation, then it must be the case that either the send completes normally, in which case the message sent was received at the destination process, or that the send is successfully canceled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive completes normally, or that the receive is successfully canceled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

If the operation has been canceled, then information to that effect will be returned in the status argument of the operation that completes the communication.

Rationale. Although the IN request handle parameter should not need to be passed by reference, the C binding has listed the argument type as MPI_Request* since MPI-1.0. This function signature therefore cannot be changed without breaking existing MPI applications. (*End of rationale.*)

MPI_TEST_CANCELLED(status, flag)

IN	status	status object (Status)
OUT	flag	(logical)

```
int MPI_Test_cancelled(const MPI_Status *status, int *flag)
MPI_Test_cancelled(status, flag, ierror) BIND(C)
    TYPE(MPI_Status), INTENT(IN) :: status
    LOGICAL, INTENT(OUT) :: flag
```

Unofficial Draft for Comment Only

1

2 3

4 5

6

 $\overline{7}$

8

9

10

11

12 13

14

15

16

17

18

19

20 21

22

23

 24

25

26

27

28

29

30

 31

32 33

34

35

40 41

1 2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
4	LOGICAL FLAG
5	INTEGER STATUS(MPI_STATUS_SIZE), IERROR
6 7	<pre>{bool MPI::Status::Is_cancelled() const(binding deprecated, see Section 15.2) }</pre>
8 9 10 11 12	Returns $flag = true$ if the communication associated with the status object was canceled successfully. In such a case, all other fields of status (such as count or tag) are undefined. Returns $flag = false$, otherwise. If a receive operation might be canceled then one should call MPI_TEST_CANCELLED first, to check whether the operation was canceled, before checking on the other fields of the return status.
13 14 15	Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (End of advice to users.)
16 17	Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation
18	of this send may require communication with the intended receiver in order to free
19	allocated buffers. On some systems this may require an interrupt to the intended
20 21	receiver. Note that, while communication may be needed to implement
21	MPI_CANCEL, this is still a local operation, since its completion does not depend on
23	the code executed by other processes. If processing is required on another process,
24	this should be transparent to the application (hence the need for an interrupt and an interrupt hendlor) (End of advice to implementance)
25	interrupt handler). (End of advice to implementors.)
26 27 28	3.9 Persistent Communication Requests
29	Often a communication with the same argument list is repeatedly executed within the in-
30	ner loop of a parallel computation. In such a situation, it may be possible to optimize
31	the communication by binding the list of communication arguments to a persistent com-
32	munication request once and, then, repeatedly using the request to initiate and complete
33	messages. The persistent request thus created can be thought of as a communication port or
34	a "half-channel." It does not provide the full functionality of a conventional channel, since
35	there is no binding of the send port to the receive port. This construct allows reduction of the every head for communication between the process and communication controller, but
36	of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another.
37	It is not necessary that messages sent with a persistent request be received by a receive
38 39	operation using a persistent request, or vice versa.
40	A persistent communication request is created using one of the five following calls.
41	These calls involve no communication.

MPI_SENI	D_INIT(buf, count, datatype, d	est, tag, comm, request)	1
IN	buf	initial address of send buffer (choice)	2
IN	count	number of elements sent (non-negative integer)	3
IN	datatype	type of each element (handle)	5
IN	dest	rank of destination (integer)	6
			7
IN	tag	message tag (integer)	8
IN	comm	communicator (handle)	9 10
OUT	request	communication request (handle)	11
	int dest, int tag, M	, int count, MPI_Datatype datatype, MPI_Comm comm, MPI_Request *request)	$^{12}_{13}$ ticket140. 14 ticket229.2.
MP1_Send_	_init(bur, count, datatyp BIND(C)	e, dest, tag, comm, request, ierror)	$^{15}_{16}$ ticket-248T.
TYPE		T(IN), ASYNCHRONOUS :: buf	17
	GER, INTENT(IN) :: count	-	18
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	19
	(MPI_Comm), INTENT(IN) ::		20
	(MPI_Request), INTENT(OUT	-	21 22
INTEC	GER, OPTIONAL, INTENT(OUT) :: lerror	23
		E, DEST, TAG, COMM, REQUEST, IERROR)	24
• -	> BUF(*)		25
INTEC	JER LREQUEST, JCOUNT, DAT	ATYPE, DEST, TAG, COMM, REQUEST, IERROR	$_{26}$ ticket250-V.
$\{MPI::Pre$	-	it(const void* buf, int count, const	27
		cype, int dest, int tag) const(binding	28 29
	deprecated, see Section	15.2) }	30
Creat	es a persistent communication	n request for a standard mode send operation, and	31
binds to it	all the arguments of a send of	operation.	32
			33
MPI_BSEN	ND_INIT(buf, count, datatype,	dest, tag, comm, request)	34
IN	buf	initial address of send buffer (choice)	35 36
IN	count	number of elements sent (non-negative integer)	37
			38
IN	datatype	type of each element (handle)	39
IN	dest	rank of destination (integer)	40
IN	tag	message tag (integer)	41 42
IN	comm	communicator (handle)	42
OUT	request	communication request (handle)	44
			45
int MPI_B		f, int count, MPI_Datatype datatype,	46 ticket 140.
	int dest, int tag, M	<pre>IPI_Comm comm, MPI_Request *request)</pre>	⁴⁷ ₄₈ ticket229.2. ticket-248T.

81

1	MPI_Bsen	nd_init(buf, com BIND(C)	unt, datatype, dest, tag, comm, request, ierror)
3 4 5 6 7 8	INTE TYPE TYPE TYPE	C(*), DIMENSION GER, INTENT(IN C(MPI_Datatype) C(MPI_Comm), IN C(MPI_Request),	<pre>(), INTENT(IN), ASYNCHRONOUS :: buf () :: count, dest, tag , INTENT(IN) :: datatype ITENT(IN) :: comm INTENT(OUT) :: request INTENT(OUT) :: ierror</pre>
9 10 ticket250-V. $\frac{11}{12}$	<typ< td=""><td>e> BUF(*)</td><td>UNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)]COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</td></typ<>	e> BUF(*)	UNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)]COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
13 14 15 16	{MPI::Pr	MPI::Data	<pre>mm::Bsend_init(const void* buf, int count, const atype& datatype, int dest, int tag) const(binding b, see Section 15.2) }</pre>
16 17 18	Crea	tes a persistent co	ommunication request for a buffered mode send.
19	MPI_SSE	ND_INIT(buf, cou	unt, datatype, dest, tag, comm, request)
20 21	IN	buf	initial address of send buffer (choice)
22	IN	count	number of elements sent (non-negative integer)
23	IN	datatype	type of each element (handle)
24 25	IN	dest	rank of destination (integer)
26	IN	tag	message tag (integer)
27	IN	comm	communicator (handle)
28 29	OUT	request	communication request (handle)
30 ticket140. ³¹ ticket229.2. ³²	int MPI_		est void* buf, int count, MPI_Datatype datatype, , int tag, MPI_Comm comm, MPI_Request *request)
ticket-248T. 34 35 36 37 38 39 40 41 42 43	TYPE INTE TYPE TYPE INTE MPI_SSEN <typ< td=""><td>BIND(C) C(*), DIMENSION CGER, INTENT(IN C(MPI_Datatype) C(MPI_Comm), IN C(MPI_Request), CGER, OPTIONAL, D_INIT(BUF, CO pe> BUF(*)</td><td><pre>punt, datatype, dest, tag, comm, request, ierror) (), INTENT(IN), ASYNCHRONOUS :: buf () :: count, dest, tag , INTENT(IN) :: datatype TTENT(IN) :: comm INTENT(OUT) :: request INTENT(OUT) :: ierror UNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) </pre></td></typ<>	BIND(C) C(*), DIMENSION CGER, INTENT(IN C(MPI_Datatype) C(MPI_Comm), IN C(MPI_Request), CGER, OPTIONAL, D_INIT(BUF, CO pe> BUF(*)	<pre>punt, datatype, dest, tag, comm, request, ierror) (), INTENT(IN), ASYNCHRONOUS :: buf () :: count, dest, tag , INTENT(IN) :: datatype TTENT(IN) :: comm INTENT(OUT) :: request INTENT(OUT) :: ierror UNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) </pre>
44 45 46 47 48		request MPI::Con MPI::Data	<pre>TATYPE, DEST, TAG, COMM, REQUEST, IERROR mm::Ssend_init(const void* buf, int count, const atype& datatype, int dest, int tag) const(binding d, see Section 15.2) }</pre>

Creat	es a persistent communication	n object for a synchronous mode send operation.	1
			2 3
MPI_RSEI	ND_INIT(buf, count, datatype,	dest, tag, comm, request)	4
IN	buf	initial address of send buffer (choice)	5
IN	count	number of elements sent (non-negative integer)	6
IN	datatype	type of each element (handle)	7
			8
IN	dest	rank of destination (integer)	10
IN	tag	message tag (integer)	11
IN	comm	communicator (handle)	12
OUT	request	communication request (handle)	13
			14
int MPI_1		f, int count, MPI_Datatype datatype,	$^{15}_{16}$ ticket 140.
	int dest, int tag, M	<pre>MPI_Comm comm, MPI_Request *request)</pre>	¹⁷ ticket229.2.
MPI_Rsen	•	pe, dest, tag, comm, request, ierror)	18 ticket-248T.
TUDE	BIND(C)		19
	(*), DIMENSION(), INTEN GER, INTENT(IN) :: count	T(IN), ASYNCHRONOUS :: buf	20 21
	(MPI_Datatype), INTENT(IN		22
	(MPI_Comm), INTENT(IN) ::	• •	23
	(MPI_Request), INTENT(OUT	-	24
INTE	GER, OPTIONAL, INTENT(OUT	') :: ierror	25
		PE, DEST, TAG, COMM, REQUEST, IERROR)	26
	e> BUF(*)		27 28
INTE	GER COUNT, DATATYPE, DEST	', TAG, COMM, REQUEST, IERROR	29
$\{MPI::Pre$	equest MPI:::Comm::Rsend_i	<pre>nit(const void* buf, int count, const</pre>	30
	• -	type, int dest, int tag) const(binding	31
	deprecated, see Section	15.2) }	32
Creat	es a persistent communication	n object for a ready mode send operation.	33 34
			35
MPI_REC	V_INIT(buf, count, datatype, s	ource, tag, comm, request)	36
OUT	buf	initial address of receive buffer (choice)	37
IN	count	number of elements received (non-negative integer)	38 39
IN	datatype	type of each element (handle)	40
		·- · · · · · · · · · · · · · · · · · ·	41
IN	source	rank of source or MPI_ANY_SOURCE (integer)	42
IN	tag	message tag or MPI_ANY_TAG (integer)	43
IN	comm	communicator (handle)	44 45
OUT	request	communication request (handle)	45 46
			47
			48

1 int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source, 2 int tag, MPI_Comm comm, MPI_Request *request) ticket-248T. 3 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) 4 BIND(C) 5TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 6 INTEGER, INTENT(IN) :: count, source, tag 7 TYPE(MPI_Datatype), INTENT(IN) :: datatype 8 TYPE(MPI_Comm), INTENT(IN) :: comm 9 TYPE(MPI_Request), INTENT(OUT) :: request 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 12MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 13 <type> BUF(*) 14INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 15{MPI::Prequest MPI::Comm::Recv_init(void* buf, int count, const 16 MPI::Datatype& datatype, int source, int tag) const(binding 17deprecated, see Section 15.2 } 18 19Creates a persistent communication request for a receive operation. The argument buf 20is marked as OUT because the user gives permission to write on the receive buffer by passing 21the argument to MPI_RECV_INIT. 22 A persistent communication request is inactive after it was created — no active com-23munication is attached to the request. 24 A communication (send or receive) that uses a persistent request is initiated by the 25function MPI_START. 2627MPI_START(request) 2829INOUT request communication request (handle) 30 31 int MPI_Start(MPI_Request *request) ticket-248T. 32 33 MPI_Start(request, ierror) BIND(C) 34 TYPE(MPI_Request), INTENT(INOUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 MPI_START(REQUEST, IERROR) 37 INTEGER REQUEST, IERROR 38 39 {void MPI::Prequest::Start() (binding deprecated, see Section 15.2) } 40The argument, request, is a handle returned by one of the previous five calls. The 41 associated request should be inactive. The request becomes active once the call is made. 42If the request is for a send with ready mode, then a matching receive should be posted 43 before the call is made. The communication buffer should not be modified after the call, 44 and until the operation completes. 45 The call is local, with similar semantics to the nonblocking communication operations 46 described in Section 3.7. That is, a call to MPI_START with a request created by 47

⁴⁸ MPI_SEND_INIT starts a communication in the same manner as a call to MPI_ISEND; a

call to MPI_START with a request created by MPI_BSEND_INIT starts a communication in the same manner as a call to MPI_IBSEND; and so on.

			4
MPI_STAF	RTALL(count, array_of_request	rs)	5
IN	count	list length (non-negative integer)	6
INOUT	array_of_requests	array of requests (array of handle)	7
			8
int MPI_S	Startall(int count, MPI_R	<pre>equest [*]array_of_requests[])</pre>	$^{9}_{10}$ ticket125. 11 ticket125.
MPI_Start	call(count, array_of_requ	ests, ierror) BIND(C)	$^{11}_{12}$ ticket-248T.
INTEC	GER, INTENT(IN) :: count		
TYPE	(MPI_Request), INTENT(INO	UT) :: array_of_requests(count)	13
INTEC	ER, OPTIONAL, INTENT(OUT) :: ierror	14
MPT STAR	TALL(COUNT, ARRAY_OF_REQU	ESTS TERROR)	15
	GER COUNT, ARRAY_OF_REQUE		16 17
			17
$\{$ static x	void MPI::Prequest::Start		18
		of_requests[])(binding deprecated, see	20
	Section 15.2 }		20
~			21

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to MPI_START (&array_of_requests[i]), executed for i=0,..., count-1, in some arbitrary order.

A communication started with a call to MPI_START or MPI_STARTALL is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL call.

A persistent request is deallocated by a call to MPI_REQUEST_FREE (Section 3.7.3).

The call to MPI_REQUEST_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes inactive. Active receive requests should not be freed. Otherwise, it will not be possible to check that the receive has completed. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

Create (Start Complete)* Free

where * indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with MPI_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI_START can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in [subsections "Problems

Unofficial Draft for Comment Only

1

2

3 4

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36 37

38 39

40

41

42

43

44

 $45 \\ 46$

47

48 ticket238-J.

Due to Data Copying and Sequence Association," and "A Problem with Register Optimization and Temporary Memory Modifications" in Section 16.2.10 on pages 673 and 679.]Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 673-676 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 16.2.16 to 16.2.19 on pages 679 to 688 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*End of advice to users.*)

3.10 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

3	MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype,	
)	source, recvtag, comm, status)	

30 31	IN	sendbuf	initial address of send buffer (choice)
32 33	IN	sendcount	number of elements in send buffer (non-negative integer)
34	IN	sendtype	type of elements in send buffer (handle)
35	IN	dest	rank of destination (integer)
36 37	IN	sendtag	send tag (integer)
38	OUT	recvbuf	initial address of receive buffer (choice)
39 40	IN	recvcount	number of elements in receive buffer (non-negative in-teger)
41 42	IN	recvtype	type of elements in receive buffer (handle)
43	IN	source	rank of source or $MPI_ANY_SOURCE\xspace$ (integer)
44	IN	recvtag	receive tag or MPI_ANY_TAG (integer)
45	IN	comm	communicator (handle)
46 47	OUT	status	status object (Status)
48			

ticket236-H. 4 ticket238-J. ⁵

ticket 238-J. 2

ticket238-J.³

1

6

7

8

9 10

11 12

13

14

15

16

17

18

19

20

21

22

23

 24

25

<pre>int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>	1 ticket 140.
int dest, int sendtag, void *recvbuf, int recvcount,	2
MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,	3
MPI_Status *status)	4
NDT Conductor (conduct conduct conduct conduct	$_5$ ticket-248T.
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,	6
recvcount, recvtype, source, recvtag, comm, status, ierror)	7
BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	8
TYPE(*), DIMENSION(), INTENT(IN) Sendbur TYPE(*), DIMENSION() :: recvbuf	9
INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,	10
-	11
recvtag	12
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	13
TYPE(MPI_Comm), INTENT(IN) :: comm	14
TYPE(MPI_Status) :: status	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,	17
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)	18
<type> SENDBUF(*), RECVBUF(*)</type>	19
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,	20
SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	21
(woid MDT, Commer Conducty (const woid toondbuf int condecunt const	22
{void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const	23
MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,	24
int recvcount, const MPI::Datatype& recvtype, int source,	25
<pre>int recvtag, MPI::Status& status) const(binding deprecated, see Section 15.2) }</pre>	26
Section 15.2	27
<pre>{void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const</pre>	28
<pre>MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,</pre>	29
<pre>int recvcount, const MPI::Datatype& recvtype, int source,</pre>	30
<pre>int recvtag) const(binding deprecated, see Section 15.2) }</pre>	31
Execute a blacking and and receive execution. Both and and receive use the same	32
Execute a blocking send and receive operation. Both send and receive use the same	33
communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.	34
	35
The semantics of a send-receive operation is what would be obtained if the caller forked	36
two concurrent threads, one to execute the send, and one to execute the receive, followed	37
by a join of these two threads.	38
	39
	40
	41

1		DRECV_REPLACE(buf, tus)	count, datatype, dest, sendtag, source, recvtag, comm, sta-
3	INOUT	buf	initial address of send and receive buffer (choice)
5	IN	count	number of elements in send and receive buffer (non-negative integer)
7	IN	datatype	type of elements in send and receive buffer (handle)
8	IN	dest	rank of destination (integer)
9	INI	sendtag	send message tag (integer)
11	IN	source	rank of source or MPI_ANY_SOURCE (integer)
12	IN	recvtag	receive message tag or MPI_ANY_TAG (integer)
13	3	comm	communicator (handle)
15	-	status	status object (Status)
16			
17	int MPI_	Sendrecv_replace(vo	id* buf, int count, MPI_Datatype datatype,
18			<pre>sendtag, int source, int recvtag, MPI_Comm comm,</pre>
ticket-248T. 20)	MPI_Status *st	atus)
21	MPI_Send		ount, datatype, dest, sendtag, source, recvtag,
22 23		<pre>comm, status, (*), DIMENSION()</pre>	<pre>ierror) BIND(C)</pre>
24			count, dest, sendtag, source, recvtag
25	5 TYPE	(MPI_Datatype), INT	ENT(IN) :: datatype
26		(MPI_Comm), INTENT(
27 28		(MPI_Status) :: st GER, OPTIONAL, INTE	
29	a		
30	MPI_SEND	COMM, STATUS,	OUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, IERROR)
31	<typ< td=""><td>e> BUF(*)</td><td></td></typ<>	e> BUF(*)	
32	INTE		, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
34	STAT	US(MPI_STATUS_SIZE)	, IERROR
35	s {void MP		eplace(void* buf, int count, const
36			<pre>x datatype, int dest, int sendtag, int source, IPI::Status& status) const(binding deprecated, see</pre>
37 38		Section 15.2 }	r1status status) const(omaing apprecated, see
39) Junid MD		eplace(void* buf, int count, const
40) γνοτα με		datatype, int dest, int sendtag, int source,
41			const (binding deprecated, see Section 15.2) }
43		ute a blocking send an	d receive. The same buffer is used both for the send and
44		-	age sent is replaced by the message received.
45		ica ta imamlana anterna M	dditional intermediate buffering is and define the " 1 "
46		ant. (End of advice to	Additional intermediate buffering is needed for the "replace" <i>implementors</i> .)
48			

3.11 Null Processes

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a non-circular shift done with calls to send-receive.

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI_PROC_NULL is executed then the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0. A matching probe (cf. Section 3.8.2) with MPI_PROC_NULL as source returns flag = true, message = MPI_MESSAGE_NULL, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0.

11 ticket38.

 $\overline{7}$

Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 29 and in Section 3.3 Data Type Matching and Data Conversion on page 37. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

4.1 Derived Datatypes

Up to here, all point to point communication have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shape and size. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A general datatype is an opaque object that specifies two things:

• A sequence of basic datatypes

 24

 $46 \\ 47$

• A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

8

Let

1

2

3

4

5

6

7

9

10 11

12

13

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\$$

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

$$Typesig = \{type_0, ..., type_{n-1}\}$$

¹⁴ be the associated type signature. This type map, together with a base address *buf*, specifies ¹⁵ a communication buffer: the communication buffer that consists of n entries, where the ¹⁶ *i*-th entry is at address *buf* + *disp_i* and has type *type_i*. A message assembled from such a ¹⁷ communication buffer will consist of n values, of the types defined by *Typesig*.

¹⁸ Most datatype constructors have replication count or block length arguments. Allowed ¹⁹ values are non-negative integers. If the value is zero, no elements are generated in the type ²⁰ map and there is no effect on datatype bounds or extent.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will use the send buffer defined by the base address buf and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base address buf and the general datatype.

General datatypes can be used in all send and receive operations. We discuss, in Section 4.1.11, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(int, 0)\}$, with one entry of type int and displacement zero. The other basic datatypes are similar.

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\$$

then

39 40 41

42 43 44

34

35

36 37

38

$$lb(Typemap) = \min_{j} disp_{j},$$

$$ub(Typemap) = \max_{j} (disp_{j} + sizeof(type_{j})) + \epsilon, \text{ and}$$

$$extent(Typemap) = ub(Typemap) - lb(Typemap).$$
(4.1)

⁴⁵ If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least ⁴⁶ non-negative increment needed to round extent(Typemap) to the next multiple of $\max_i k_i$. ⁴⁷ In Fortran, whether the alignments k_i are computed according to the alignments used by ⁴⁸ the compiler in common blocks, SEQUENCE derived types, BIND(C) derived types, or derived

Unofficial Draft for Comment Only

ticket229.2. 47

ticket229.2.

types that are neither SEQUENCE nor BIND(C), is implementation-dependent. The complete definition of **extent** is given on page 113 in Section 4.1.6 on page 112.

Example 4.1 Assume that $Type = \{(double, 0), (char, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, SEQUENCE derived types, or BIND(C) derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI_TYPE_CREATE_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 4.1.6 on page 112 and in Section 16.2.15 on page 677. (End of rationale.)

Type Constructors with Explicit Addresses 4.1.1

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, **MPI_TYPE_CREATE_HINDEXED_BLOCK**, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type MPI_Aint and MPI::Aint are used in C and C++. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8.

Datatype Constructors 4.1.2

Contiguous The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

	'E_CONTIGUOUS(count, oldty	pe, newtype)	35
IN	count	replication count (non-negative integer)	36
IN	oldtype	old datatype (handle)	37
	51		38
OUT	newtype	new datatype (handle)	39
			40
int MPI_	Type_contiguous(int count	;, MPI Datatype oldtype,	41
		· _ · · · · · · · · · · · · · · · · · ·	
_	MPI_Datatype *newtyp		42
	MPI_Datatype *newtyp	pe)	$^{42}_{43}$ ticket-248T.
МРІ_Туре	MPI_Datatype *newtyp _contiguous(count, oldtyp	pe) pe, newtype, ierror) BIND(C)	
MPI_Type INTE	MPI_Datatype *newtyp contiguous(count, oldtyp GER, INTENT(IN) :: count	pe) pe, newtype, ierror) BIND(C)	$_{43}$ ticket-248T.
MPI_Type INTE TYPE	MPI_Datatype *newtyp _contiguous(count, oldtyp GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN	pe) pe, newtype, ierror) BIND(C) ; i) :: oldtype	43 ticket-248T.
MPI_Type INTE TYPE TYPE	MPI_Datatype *newtyp contiguous(count, oldtyp GER, INTENT(IN) :: count	pe) pe, newtype, ierror) BIND(C) ; I) :: oldtype IT) :: newtype	43 ticket-248T. 44 45

1

2

3

4

5

6

 $\overline{7}$

8 9

10

11

12

13

15

16

17

18

1920

2122

23

 24

25

26

27

2829

30

31

32 33 34

14 ticket 229.2.

ticket 280

	1 2		CONTIGUOUS(COUNT, OLDTYP ER COUNT, OLDTYPE, NEWTY	
	3 4 5	{MPI::Data	atype MPI::Datatype::Cre deprecated, see Section	<pre>ate_contiguous(int count) const(binding 15.2) }</pre>
	6 7 8			by concatenating count copies of <i>extent</i> as the size of the concatenated copies.
	9 10	-	4.2 Let oldtype have type r The type map of the datatyp	nap $\{(double, 0), (char, 8)\}$, with extent 16, and let be returned by newtype is
	11 12	{(dou	ble, 0), (char, 8), (double, 16),	$(char, 24), (double, 32), (char, 40)\};$
	13 14	i.e., alterna	ting double and char element	s, with displacements $0, 8, 16, 24, 32, 40$.
	15 16	In gene	eral, assume that the type m	ap of oldtype is
	17 18	$\{(type$	$e_0, disp_0), \dots, (type_{n-1}, disp_{n-1})$	-1)},
	19	with extent	ex. Then newtype has a type	be map with $count \cdot n$ entries defined by:
	20	{(tupe	$e_0, disp_0), \dots, (tupe_{n-1}, disp_n)$	$(type_0, disp_0 + ex),, (type_{n-1}, disp_{n-1} + ex),$
	21 22	2		
	23	,(ty	$upe_0, disp_0 + ex \cdot (count - 1))$	$,,(type_{n-1},disp_{n-1}+ex\cdot(count-1))\}.$
	24			
	25			
	26			
	27 28			OR is a more general constructor that allows repli- t consist of equally spaced blocks. Each block is
	29			imber of copies of the old datatype. The spacing
	30	ě	ocks is a multiple of the exte	
	31			
	32 33	MPI_TYPE	_VECTOR(count, blocklengt	h, stride, oldtype, newtype)
	34	IN	count	number of blocks (non-negative integer)
	35	IN	blocklength	
	36 37	IIN	biockiength	number of elements in each block (non-negative integer)
	38	IN	stride	number of elements between start of each block (inte-
	39			ger)
	40	IN	oldtype	old datatype (handle)
	41 42	OUT	newtype	new datatype (handle)
	43			
	44	int MPI_Ty	-	t blocklength, int stride,
ticket-248T.	45		MPI_Datatype oldtype	, MPI_Datatype *newtype)
	46	MPI_Type_v		h, stride, oldtype, newtype, ierror)
	47 48	ͳͶͲϾʹϹͳ	BIND(C) ER, INTENT(IN) :: count	blocklength stride
			En, INTENT(IN) :: COUNT	, procertenken, peride

TYPE(MPI_Datatype), INTENT(IN) :: oldtype	1
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3 4
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)	5
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	6
{MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength,	7
<pre>int stride) const(binding deprecated, see Section 15.2) }</pre>	8
	9 10
Example 4.3 Assume, again, that oldtype has type map $\{(double, 0), (char, 8)\}$, with extent	10
16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with	12
type map,	13
$\{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40),$	14
$(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104) \}.$	15 16
That is, two blocks with three copies each of the old type, with a stride of 4 elements $(4 \cdot 16)$	17
bytes) between the blocks.	18
Example 4.4. A subtraction $V = V = V = V = 0$	19
Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,	20 21
	22
$\{({\sf double},0),({\sf char},8),({\sf double},-32),({\sf char},-24),({\sf double},-64),({\sf char},-56)\}.$	23
	24
In general, assume that oldtype has type map,	25
$\{(type_0, disp_0),, (type_{n-1}, disp_{n-1})\},\$	26 27
with extent ex . Let bl be the blocklength. The newly created datatype has a type map with	28
count \cdot bl \cdot <i>n</i> entries:	29
$\{(type_0, disp_0),, (type_{n-1}, disp_{n-1}),$	30 31
$(type_0, disp_0 + ex),, (type_{n-1}, disp_{n-1} + ex),,$	32
$(type_0, disp_0 + (bl - 1) \cdot ex),, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$	33
	34 35
$(type_0, disp_0 + stride \cdot ex),, (type_{n-1}, disp_{n-1} + stride \cdot ex),,$	36
$(type_0, disp_0 + (stride + bl - 1) \cdot ex),, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex),,$	37
$(type_0, disp_0 + stride \cdot (count - 1) \cdot ex),,$	38 39
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex),,$	40
$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex),,$	41
	42 43
$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$	43
	45
A call to $MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)$ is equivalent to a call to	46
$MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), \mathrm{or} \ \mathrm{to} \ \mathrm{a} \ \mathrm{call} \ \mathrm{to} \ MPI_TYPE_VECTOR(1, 1, 1, oldtype, newtype), \mathrm{or} \ \mathrm{to} \ \mathrm{a} \ \mathrm{call} \ \mathrm{to} \ MPI_TYPE_VECTOR(1, 1, 1, 1, oldtype, newtype), \mathrm{or} \ \mathrm{to} \ \mathrm{a} \ \mathrm{call} \ \mathrm{to} \ MPI_TYPE_VECTOR(1, 1, 1, 1, oldtype, newtype), \mathrm{or} \ \mathrm{to} \ \mathrm{a} \ \mathrm{call} \ \mathrm{b} \ MPI_TYPE_VECTOR(1, 1, 1, 1, oldtype, newtype), \mathrm{or} \ \mathrm{to} \ \mathrm{a} \ \mathrm{call} \ \mathrm{b} \ o \ b \ a \ call \ b \ b \ b \ b \ b \ b \ b \ b \ b \ call \ b \ $	47
count, n, oldtype, newtype), n arbitrary.	48

```
1
                  Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to
            \mathbf{2}
                  MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The
            3
                  use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for
            4
                  "heterogeneous").
            5
            6
                  MPI_TYPE_CREATE_HVECTOR( count, blocklength, stride, oldtype, newtype)
            7
             8
                    IN
                                                             number of blocks (non-negative integer)
                               count
            9
                    IN
                               blocklength
                                                             number of elements in each block (non-negative inte-
            10
                                                             ger)
            11
                    IN
                               stride
                                                             number of bytes between start of each block (integer)
            12
            13
                    IN
                               oldtype
                                                             old datatype (handle)
            14
                    OUT
                               newtype
                                                             new datatype (handle)
            15
            16
                  int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
            17
                                  MPI_Datatype oldtype, MPI_Datatype *newtype)
            18
ticket-248T.
            19
                  MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
            20
                                  ierror) BIND(C)
            21
                       INTEGER, INTENT(IN) :: count, blocklength
            22
                       INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
            23
                       TYPE(MPI_Datatype), INTENT(IN) :: oldtype
            24
                       TYPE(MPI_Datatype), INTENT(OUT) :: newtype
            25
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            26
                  MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
            27
                                   IERROR)
            28
                       INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
            29
                       INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
            30
            31
                  {MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
            32
                                  MPI:::Aint stride) const(binding deprecated, see Section 15.2) }
            33
                       This function replaces MPI_TYPE_HVECTOR, whose use is deprecated. See also Chap-
            34
                  ter 15.
            35
            36
            37
                       Assume that oldtype has type map,
            38
                        \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\
            39
            40
                  with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
            41
                  count \cdot bl \cdot n entries:
            42
                        \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}), \}
            43
            44
                        (type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,
            45
            46
                        (type_0, disp_0 + (bl - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
            47
            48
                        (type_0, disp_0 + \mathsf{stride}), \dots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \dots,
```

$(type_0, disp_0 + stride + (bl - 1) \cdot ex),,$	1
	2
$(type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex),,$	3
	4
$(type_0, disp_0 + stride \cdot (count - 1)),, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)),,$	5
$(type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex),,$	6 7
	8
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.$	9
	10

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

	18				
MPI_TYP	19				
	type)		20		
IN	count	number of blocks – also number of entries in	21		
		$array_of_displacements and array_of_blocklengths (non-$	22		
		negative integer)	23		
IN	array_of_blocklengths	number of elements per block (array of non-negative	24		
ii v	andy_or_brocklengths	integers)	25		
			26		
IN	array_of_displacements	displacement for each block, in multiples of oldtype	27		
		extent (array of integer)	28		
IN	oldtype	old datatype (handle)	29		
OUT	newtype	new datatype (handle)	30		
001	newtype		31		
		and int warman of blacklangths count	32		
int MPI_		<pre>const int *array_of_blocklengths, const </pre>	$_{33}$ ticket140. $_{34}$ ticket140.		
	MPI_Datatype *newtyp	acements, MPI_Datatype oldtype,	$_{34}$ 010KC0140.		
	MFI_Datatype *newtyp		35 ticket-248T.		
MPI_Type	_indexed(count, array_of_	blocklengths, array_of_displacements,	36		
	oldtype, newtype, ie	error) BIND(C)	37		
INTE	GER, INTENT(IN) :: count	<pre>, array_of_blocklengths(count),</pre>	38		
arra	y_of_displacements(count)		39		
TYPE	(MPI_Datatype), INTENT(IN) :: oldtype	40		
TYPE	(MPI_Datatype), INTENT(OU	T) :: newtype	41		
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	42		
MDT TVDE	INDEXED (COUNT ADDAY OF	BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,	43		
MPI_IIPE	44				
ΤΝΤΡΓΡ	45				
	INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),				
OLDTYPE, NEWTYPE, IERROR 47					

 $14 \\ 15$

```
1
       {MPI::Datatype MPI::Datatype::Create_indexed(int count,
\mathbf{2}
                          const int array_of_blocklengths[],
3
                          const int array_of_displacements[]) const(binding deprecated, see
4
                          Section 15.2 }
5
6
       Example 4.5
7
            Let oldtype have type map {(double, 0), (char, 8)}, with extent 16. Let B = (3, 1) and
8
       let D = (4, 0). A call to MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) returns a datatype
9
       with type map,
10
11
              {(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104),
12
              (\mathsf{double}, 0), (\mathsf{char}, 8)\}.
13
14
       That is, three copies of the old type starting at displacement 64, and one copy starting at
15
       displacement 0.
16
17
18
            In general, assume that oldtype has type map,
19
              \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\
20
21
                                 Let B be the array_of_blocklength argument and D be the
       with extent ex.
22
       array_of_displacements argument. The newly created datatype has n \cdot \sum_{i=0}^{\text{count}-1} B[i] entries:
23
^{24}
              \{(type_0, disp_0 + D[0] \cdot ex), ..., (type_{n-1}, disp_{n-1} + D[0] \cdot ex), ..., \}
25
26
              (type_0, disp_0 + (D[0] + B[0] - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), ...,
27
              (type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] \cdot ex), ...,
28
29
              (type_0, disp_0 + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,
30
^{31}
              (type_{n-1}, disp_{n-1} + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.
32
33
34
35
             A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent
36
       to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where
37
              D[i] = j \cdot \text{stride}, \ j = 0, ..., \text{count} - 1,
38
39
       and
40
41
              B[j] = blocklength, j = 0, ..., count - 1.
42
43
44
45
46
47
48
```

Hindexed	The function MPI_TYPE_C	REATE_HINDEXED is identical to	1	
MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are spec-			2	
ified in bytes, rather than in multiples of the oldtype extent.			3	
			4 5	
MPI_TYP	'E_CREATE_HINDEXED(coun	t, array_of_blocklengths, array_of_displacements, old-	6	
	type, newtype)		7	
IN	count	number of blocks — also number of entries in	8	
		array_of_displacements and array_of_blocklengths (non-	9	
		negative integer)	10	
IN	array_of_blocklengths	number of elements in each block (array of non-negative	11	
		integers)	12	
IN	array_of_displacements	byte displacement of each block (array of integer)	13	
			14 15	
IN	oldtype	old datatype (handle)	16	
OUT	newtype	new datatype (handle)	17	
			18	
int MPI_		<pre>count, const int array_of_blocklengths[],</pre>	$_{19}$ ticket 140.	
	-	<pre>y_of_displacements[], MPI_Datatype oldtype,</pre>	$_{20}$ ticket 140.	
	MPI_Datatype *newtyp	pe)	21 ticket-248T.	
MPI_Type	_create_hindexed(count, a	rray_of_blocklengths,	22	
	• -	nts, oldtype, newtype, ierror) BIND(C)	23	
		<pre>array_of_blocklengths(count)</pre>	24	
	GER(KIND=MPI_ADDRESS_KIND		25 26	
	y_of_displacements(count)		27	
	(MPI_Datatype), INTENT(IN (MPI_Datatype), INTENT(OU		28	
	GER, OPTIONAL, INTENT(OUT	• •	29	
			30	
MPI_TYPE	CREATE_HINDEXED(COUNT, A		31	
T.)(010		VTS, OLDTYPE, NEWTYPE, IERROR)	32	
		<pre>XLENGTHS(*), OLDTYPE, NEWTYPE, IERROR) ARRAY_OF_DISPLACEMENTS(*)</pre>	33	
	GER (KIND-MFI_ADDRESS_KIND) ARRAI_OF_DISPLACEMENTS(*)	34	
{MPI::Da	tatype MPI::Datatype::Cre		35	
	const int array_of_b	-	36 37	
		ay_of_displacements[]) const(binding	38	
	deprecated, see Section	15.2) }	39	
This	function replaces MPI_TYPE_1	HINDEXED, whose use is deprecated. See also Chap-	40	
ter 15 .			41	
			42	
Assume that oldtype has type map,				
			44	
$\{(type_0, disp_0),, (type_{n-1}, disp_{n-1})\},\$			45 46	
with exte	with extent ex . Let B be the array_of_blocklength argument and D be the			
arrav of o	47			

<code>array_of_displacements</code> argument. The newly created datatype has a type map with n ·

1	$\sum_{i=0}^{\text{count}-1} B[i]$ entries:				
2 3	$\{(ty)$	$pe_0, disp_0 + D[0]),, (type_n - $	$_{-1}, disp_{n-1} + D[0]),,$		
4 5	$(type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex),,$				
6 7	$(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex),,$				
8	$(type_0, disp_0 + D[count-1]),, (type_{n-1}, disp_{n-1} + D[count-1]),,$				
9 10	$(type_0, disp_0 + D[count-1] + (B[count-1] - 1) \cdot ex),,$				
11 12	(typ	$e_{n-1}, disp_{n-1} + D[count-1] +$	- $(B[count-1]-1) \cdot ex)$.		
12					
14 15					
16	Indexed_b	lock This function is the sa	ame as $MPI_TYPE_INDEXED$ except that the block-		
17 18	0		ere are many codes using indirect addressing arising locksize is always 1 (gather/scatter). The following		
19		0	ant blocksize and arbitrary displacements.		
20					
21 22	MPI_TYP		CK(count, blocklength, array_of_displacements, oldtype,		
23		newtype)			
24 25	IN	count	length of array of displacements (non-negative integer)		
26	IN	blocklength	size of block (non-negative integer)		
27	IN	array_of_displacements	array of displacements (array of integer)		
28	IN	oldtype	old datatype (handle)		
29 30	OUT	newtype	new datatype (handle)		
ticket140. $\frac{31}{32}$ 33 ticket-248T. 34	<pre>int MPI_Type_create_indexed_block(int count, int blocklength, const</pre>				
35 36 37 38 39 40 41 42	<pre>MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,</pre>				
43 44 45 46	MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR				
47 48	{MPI::Da	tatype MPI::Datatype::Cn int blocklength,	reate_indexed_block(int count,		

	<pre>const int array_o: Section 15.2) }</pre>	<pre>f_displacements[]) const(binding deprecated, see</pre>	1 2 3 ticket280.
			4
		TYPE_CREATE_HINDEXED_BLOCK is identical to	5
		OCK, except that block displacements in	6
array_o	- displacements are specified if	n bytes, rather than in multiples of the oldtype extent.	7
			8
MPI_T	PE_CREATE_HINDEXED_BI	LOCK(count, blocklength, array_of_displacements, old-	9 10
	type, newtype)		11
IN	count	length of array of displacements (non-negative integer)	12
IN	blocklength	size of block (non-negative integer)	13
IN	array_of_displacements	byte displacement of each block (array of integer)	14
	· · ·		15
IN	oldtype	old datatype (handle)	16
OUT	newtype	new datatype (handle)	17
			18
int MP	I_Type_create_hindexed_b]	lock(int count, int blocklength,	19
	MPI_Aint array_of	_displacements[], MPI_Datatype oldtype,	20 21
	MPI_Datatype *new	type)	21 22 ticket-248T.
MPI Tv	pe create hindexed block((count, blocklength, array_of_displacements,	²² UICKet-2401.
J	oldtype, newtype,		24
IN	TEGER, INTENT(IN) :: cou		25
IN	TEGER(KIND=MPI_Address_ki	ind), INTENT(IN) ::	26
ar	ray_of_displacements(cour	nt)	27
TY	PE(MPI_Datatype), INTENT((IN) :: oldtype	28
	PE(MPI_Datatype), INTENT(• •	29
IN	TEGER, OPTIONAL, INTENT(C	JUT) :: ierror	30
MPI_TY	PE_CREATE_HINDEXED_BLOCK((COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	31
_	OLDTYPE, NEWTYPE,		32
IN	TEGER COUNT, BLOCKLENGTH,	, OLDTYPE, NEWTYPE, IERROR	33
IN	TEGER(KIND=MPI_ADDRESS_KI	IND) ARRAY_OF_DISPLACEMENTS(*)	34
			35 36
Cturret	MDI TYPE STRUCT is the	most monoral time constructor. It further monoralized	37
		most general type constructor. It further generalizes that it allows each block to consist of replications of	38
	t datatypes.	that it anows each block to consist of replications of	39
unicien	t datatypes.		40
			41
42			
			43
			44

```
1
                 MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements,
            \mathbf{2}
                                array_of_types, newtype)
            3
                   IN
                                                         number of blocks (non-negative integer) — also num-
                             count
            4
                                                         ber of entries in arrays array_of_types,
            5
                                                         array_of_displacements and array_of_blocklengths
            6
                   IN
                             array_of_blocklength
                                                         number of elements in each block (array of non-negative
            7
                                                         integer)
            8
            9
                   IN
                             array_of_displacements
                                                         byte displacement of each block (array of integer)
            10
                   IN
                             array_of_types
                                                         type of elements in each block (array of handles to
            11
                                                         datatype objects)
           12
                   OUT
                                                         new datatype (handle)
                             newtype
           13
           14
                 int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
  ticket140. 15
  ticket140. 16
                                const MPI_Aint array_of_displacements[], const
  ticket140.17
                                MPI_Datatype array_of_types[], MPI_Datatype *newtype)
ticket-248T. 18
                 MPI_Type_create_struct(count, array_of_blocklengths,
            19
                                array_of_displacements, array_of_types, newtype, ierror)
           20
                                BIND(C)
           21
                      INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
           22
                      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
           23
                      array_of_displacements(count)
           24
                      TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
           25
                      TYPE(MPI_Datatype), INTENT(OUT) :: newtype
            26
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           27
                 MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
           28
                                 ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
           29
           30
                      INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
           31
                      IERROR
           32
                      INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
           33
                 {static MPI::Datatype MPI::Datatype::Create_struct(int count,
           34
                                const int array_of_blocklengths[], const MPI::Aint
           35
                                array_of_displacements[],
           36
                                const MPI::Datatype array_of_types[])(binding deprecated, see
           37
                                 Section 15.2 }
           38
           39
                     This function replaces MPI_TYPE_STRUCT, whose use is deprecated. See also Chap-
           40
                 ter 15.
           41
                 Example 4.6 Let type1 have type map,
           42
           43
                       \{(double, 0), (char, 8)\},\
           44
           45
                 with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR).
           46
                 Then a call to MPI_TYPE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,
            47
                       \{(float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)\}.
            48
```

That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)

In general, let T be the array_of_types argument, where T[i] is a handle to,

$$typemap_{i} = \{(type_{0}^{i}, disp_{0}^{i}), ..., (type_{n_{i}-1}^{i}, disp_{n_{i}-1}^{i})\},\$$

with extent ex_i . Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{c-1} B[i] \cdot n_i$ entries:

$$\{(type_0^0, disp_0^0 + D[0]), ..., (type_{n_0}^0, disp_{n_0}^0 + D[0]), ..., \\ (type_0^0, disp_0^0 + D[0] + (B[0] - 1) \cdot ex_0), ..., (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), ..., \\ (type_0^{c-1}, disp_0^{c-1} + D[c-1]), ..., (type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1]), ..., \\ (type_0^{c-1}, disp_0^{c-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{c-1}), ..., \\ (type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{c-1})\}.$$

A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T is equal to oldtype.

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN	ndims	number of array dimensions (positive integer)	33
IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)	34 35 36
IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)	36 37 38
IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)	39 40
IN	order	array storage order flag (state)	41
IN	oldtype	array element datatype (handle)	42 43
OUT	newtype	new datatype (handle)	44

int MPI_Type_create_subarray(int ndims, const int array_of_sizes[], const int array_of_subsizes[], const int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

Unofficial Draft for Comment Only

 24

 31

ticket-248T.

```
1
2
     MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                     array_of_starts, order, oldtype, newtype, ierror) BIND(C)
3
4
          INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
          array_of_subsizes(ndims), array_of_starts(ndims), order
5
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
6
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
7
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
10
                     ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
11
          INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
12
          ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
13
14
     {MPI::Datatype MPI::Datatype::Create_subarray(int ndims,
                     const int array_of_sizes[], const int array_of_subsizes[],
15
16
                     const int array_of_starts[], int order) const/binding deprecated.
                     see Section 15.2 }
17
18
          The subarray type constructor creates an MPI datatype describing an n-dimensional
19
     subarray of an n-dimensional array. The subarray may be situated anywhere within the
20
     full array, and may be of any nonzero size up to the size of the larger array as long as it
21
     is confined within this array. This type constructor facilitates creating filetypes to access
22
     arrays distributed in blocks among processes to a single file that contains the global array,
23
     see MPI I/O, especially Section 13.1.1 on page 511.
24
          This type constructor can handle arrays with an arbitrary number of dimensions and
25
     works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note
26
     that a C program may use Fortran order and a Fortran program may use C order.
27
          The ndims parameter specifies the number of dimensions in the full data array and
28
     gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.
29
          The number of elements of type oldtype in each dimension of the n-dimensional ar-
30
     ray and the requested subarray are specified by array_of_sizes and array_of_subsizes, re-
^{31}
     spectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] < 1 or
32
     array_of_subsizes[i] > array_of_sizes[i].
33
          The array_of_starts contains the starting coordinates of each dimension of the subarray.
34
     Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to
35
     specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).
36
37
           Advice to users. In a Fortran program with arrays indexed starting from 1, if the
38
           starting coordinate of a particular dimension of the subarray is n, then the entry in
           array_of_starts for that dimension is n-1. (End of advice to users.)
39
40
          The order argument specifies the storage order for the subarray as well as the full array.
41
     It must be set to one of the following:
42
     MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)
43
44
     MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)
45
          A ndims-dimensional subarray (newtype) with no extra padding can be defined by the
46
     function Subarray() as follows:
47
48
           newtype = Subarray(ndims, {size_0, size_1, \ldots, size_{ndims-1}},
```

Let the typemap of **oldtype** have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when $order = MPI_ORDER_FORTRAN$, and Equation 4.4 defines the recursion step when $order = MPI_ORDER_C$.

$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \}$	(4.2)	11
	(4.2)	1:
$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$		1
$= \{(MPI_LB, 0),$		14
$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$		10
$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$		1'
$disp_{n-1} + (start_0 + 1) \times ex), \dots$		18
$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,$		19
$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$		20
$(MPI_UB, size_0 \times ex)$		21
		22 23
Subarray($ndims$, { $size_0, size_1, \ldots, size_{ndims-1}$ },	(4.3)	24
$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	(10)	25
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$		26
		27
$= \text{Subarray}(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},$		28
$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$		29
$\{start_1, start_2, \dots, start_{ndims-1}\},\$		30
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$		31 32
Subarray($ndims$, { $size_0, size_1, \ldots, size_{ndims-1}$ },	(4.4)	3: 34
$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$		35
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$		36
$= \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$		37
$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},\$		38
$\{start_0, start_1, \dots, start_{ndims-2}\},\$		39
	$dt_{i}(\mathbf{p}, \mathbf{q}^{i})$	40
Subarray(1, $\{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, ol$		4
an example use of $MPI_TYPE_CREATE_SUBARRAY$ in the context of $\mathrm{I/O}$	see Sec-	43

tion 13.9.2.

4546

44

4748

Distributed Array Datatype Constructor 4.1.4

The distributed array type constructor supports HPF-like [42] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4 56

 $\overline{7}$

8

9

1 2 3 4 5 6 7 8 9 10 11	foll this set to on ope adv	lows. Complementary file s constructor with identic appropriately). These fil define the view (via MPI_ page 511 and Section 13 eration (with identical off vice to users.)	reate an HPF-like file view using this type constructor as stypes are created by having every process of a group call cal arguments (with the exception of rank which should be etypes (along with identical disp and etype) are then used FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1 .3 on page 524. Using this view, a collective data access fsets) will yield an HPF-like distribution pattern. (<i>End of</i>	
12	MPI_TY		ze, rank, ndims, array_of_gsizes, array_of_distribs, ray_of_psizes, order, oldtype, newtype)	
13 14	IN	size	size of process group (positive integer)	
15	IN	rank	rank in process group (non-negative integer)	
16 17 18	IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)	
19 20	IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)	
21	IN	array_of_distribs	distribution of array in each dimension (array of state)	
22 23 24	IN	array_of_dargs	distribution argument in each dimension (array of pos- itive integers)	
25 26	IN	array_of_psizes	size of process grid in each dimension (array of positive integers)	
27	IN	order	array storage order flag (state)	
28 29	IN	oldtype	old datatype (handle)	
30 31	OUT	newtype	new datatype (handle)	
ticket140. ³² ticket140. ³³ ticket140. ³⁴ ticket140. ³⁵ ticket-248T. ³⁶	<pre>int MPI_Type_create_darray(int size, int rank, int ndims, const</pre>			
37 38 39 40 41 42 43 44 45 46 47	<pre>MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,</pre>			
47 48		ARRAY_OF_DISTRI OLDTYPE, NEWTYF		

<pre>INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*), ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR</pre>	1 2
	3
{MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims,	4
<pre>const int array_of_gsizes[], const int array_of_distribs[],</pre>	5
<pre>const int array_of_dargs[], const int array_of_psizes[],</pre>	6
<pre>int order) const(binding deprecated, see Section 15.2) }</pre>	7
MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to	8
the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional	9
grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See	10
Example 4.7, page 110.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the	11
equation $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$ must be satisfied. The ordering of processes	12
in the process grid is assumed to be row-major, as in the case of virtual Cartesian process	13
topologies .	14
Advice to users. For both Fortran and C arrays, the ordering of processes in the	15
process grid is assumed to be row-major. This is consistent with the ordering used in	16
virtual Cartesian process topologies in MPI. To create such virtual process topologies,	17
or to find the coordinates of a process in the process grid, etc., users may use the	18
corresponding process topology functions, see Chapter 7 on page 301. (End of advice	19
to users.)	20 21
Each dimension of the array can be distributed in one of three ways:	21
	22
• MPI_DISTRIBUTE_BLOCK - Block distribution	24
• MPI_DISTRIBUTE_CYCLIC - Cyclic distribution	25
• MPI_DISTRIBUTE_NONE - Dimension not distributed.	26
The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument.	27 28
The distribution argument for a dimension that is not distributed is ignored. For any	20
dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify	30
array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].	31
For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to	32
MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-	33
$RAY(BLOCK)$ corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of	34
MPI_DISTRIBUTE_DFLT_DARG.	35
The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the stor-	36
age order. Therefore, arrays described by this type constructor may be stored in Fortran	37
(column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN	38
and MPI_ORDER_C.	39
This routine creates a new MPI datatype with a typemap defined in terms of a function	40
called "cyclic()" (see below).	41
Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.	42
MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the	43
MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.	44
MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG	45 46
is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to	40 47
$(\operatorname{array_of}_{gsizes}[i] + \operatorname{array}_{of}_{psizes}[i] - 1)/\operatorname{array}_{of}_{psizes}[i].$	48
$(a_1a_y_0_k_s_c_s_l_l + a_1a_y_0_k_s_c_s_l_l - 1)/a_1a_y_0_k_s_c_s_l_l.$	

```
1
      If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and
\mathbf{2}
      MPI_DISTRIBUTE_CYCLIC are equivalent.
3
          MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with
4
      array_of_dargs[i] set to array_of_gsizes[i].
5
          Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to
6
      MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with
7
      array_of_dargs[i] set to 1.
8
          For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined
9
     by the following code fragment:
10
          oldtype[0] = oldtype;
11
          for ( i = 0; i < ndims; i++ ) {</pre>
12
              oldtype[i+1] = cyclic(array_of_dargs[i],
13
                                        array_of_gsizes[i],
14
                                        r[i],
15
                                        array_of_psizes[i],
16
                                        oldtype[i]);
17
          }
18
19
          newtype = oldtype[ndims];
20
          For MPI_ORDER_C, the code is:
21
22
          oldtype[0] = oldtype;
23
          for ( i = 0; i < ndims; i++ ) {</pre>
24
              oldtype[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
25
                                          array_of_gsizes[ndims - i - 1],
26
                                          r[ndims - i - 1],
27
                                          array_of_psizes[ndims - i - 1],
28
                                          oldtype[i]);
29
          }
30
          newtype = oldtype[ndims];
^{31}
32
33
      where r[i] is the position of the process (with rank rank) in the process grid at dimension i.
34
      The values of r[i] are given by the following code fragment:
35
36
               t_rank = rank;
37
               t_size = 1;
38
               for (i = 0; i < ndims; i++)</pre>
39
                        t_size *= array_of_psizes[i];
40
               for (i = 0; i < ndims; i++) {</pre>
41
                    t_size = t_size / array_of_psizes[i];
42
                    r[i] = t_rank / t_size;
43
                    t_rank = t_rank % t_size;
44
               }
45
46
          Let the typemap of oldtype have the form:
47
48
           \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
```

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Given the above, the function cyclic() is defined as follows: cyclic(*darg*, *gsize*, *r*, *psize*, oldtype) $= \{(MPI_LB, 0), \}$ $(type_0, disp_0 + r \times darq \times ex), \ldots,$ $(type_{n-1}, disp_{n-1} + r \times darg \times ex),$ $(type_0, disp_0 + (r \times darg + 1) \times ex), \ldots,$ $(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex),$ 10 11 $(type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots,$ 1213 $(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),$ 1415 $(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,$ 1617 $(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$ 18 $(type_0, disp_0 + (r \times darq + 1) \times ex + psize \times darq \times ex), \dots,$ 19 $(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$ 20. . . 21 $(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$ 22 23 $(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),$ 24 25 $(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$ 2627 $(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$ 28 $(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,$ 29 $(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$ 30 $+psize \times darg \times ex \times (count - 1)),$ 3132 . . . 33 $(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$ 34 $+psize \times darg \times ex \times (count - 1)), \ldots,$ 35 $(type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex$ 36 $+psize \times darg \times ex \times (count - 1)),$ 37 38 $(MPI_UB, gsize * ex)$ 39 where *count* is defined by this code fragment: 40 41 nblocks = (gsize + (darg - 1)) / darg; 42count = nblocks / psize; 43

left_over = nblocks - count * psize; if (r < left_over)</pre> count = count + 1;

Here, *nblocks* is the number of blocks that must be distributed among the processors. Finally, $darg_{last}$ is defined by this code fragment:

Unofficial Draft for Comment Only

1 2

3

4

5

6 7

9

44

45

4647

```
1
              if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
\mathbf{2}
                   darg_last = darg;
3
              else
4
                   darg_last = num_in_last_cyclic - darg * r;
5
                   if (darg_last > darg)
6
                           darg_last = darg;
7
                   if (darg_last <= 0)
8
                           darg_last = darg;
9
10
     Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:
11
12
            <oldtype> FILEARRAY(100, 200, 300)
13
     !HPF$ PROCESSORS PROCESSES(2, 3)
14
     !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
15
     This can be achieved by the following Fortran code, assuming there will be six processes
16
     attached to the run:
17
18
         ndims = 3
19
         array_of_gsizes(1) = 100
20
         array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
21
         array_of_dargs(1) = 10
22
         array_of_gsizes(2) = 200
23
         array_of_distribs(2) = MPI_DISTRIBUTE_NONE
24
         \operatorname{array_of_dargs}(2) = 0
25
         array_of_gsizes(3) = 300
26
         array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
27
         array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
28
         array_of_psizes(1) = 2
29
         array_of_psizes(2) = 1
30
         array_of_psizes(3) = 3
31
         call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
32
         call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
33
         call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
34
               array_of_distribs, array_of_dargs, array_of_psizes,
                                                                                &
35
               MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
36
37
```

4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM.

⁴⁵ The address of a location in memory can be found by invoking the function
 ⁴⁶ MPI_GET_ADDRESS.

47 48

MPI_GET_ADDRESS(location, address)

MPI_GE	I_ADDRESS(location, a	ddress)	1
IN	location	location in caller memory (choice)	2
OUT	address	address of location (integer)	3
			5
int MPI	_Get_address(<mark>const</mark> v	oid *location, MPI_Aint *address)	⁶ ticket140.
MPT Get	address(location a	ddress, ierror) BIND(C)	7 ticket229.3.
		ASYNCHRONOUS :: location	$_{8}$ ticket-248T.
		S_KIND), INTENT(OUT) :: address	9
	EGER, OPTIONAL, INTE		10
			11 12
	_ADDRESS(LOCATION, A pe> LOCATION(*)	DDRESS, IERROR)	12
• .	EGER IERROR		14
	EGER(KIND=MPI_ADDRES	S KIND) ADDRESS	15
			16
{MPI::A:		(void* location) (binding deprecated, see Section 15.2)	17
	}		18
This	function replaces MPI_	ADDRESS, whose use is deprecated. See also Chapter 15.	19
	Irns the (byte) address	· · · ·	20
			21
		nt Fortran MPI codes will run unmodified, and will port	22
		they may fail if addresses larger than $2^{32} - 1$ are used	23
	- 0	es should be written so that they use the new functions.	24
		y with $C/C++$ and avoids errors on 64 bit architectures.	25
		en codes may need to be (slightly) rewritten to port to old	26
		hat do not support KIND declarations. (End of advice to	27 28
use	ers.)		
			29 ticket229.2.
Ra	tionale. In the mpi_	_f08 module, the location argument is not defined with	31
INT	TENT (IN) because existin	ng applications may use MPI_GET_ADDRESS (and the dep-	32
rec	ated $MPI_ADDRESS$) as	s a substitute for MPI_F_SYNC_REG that was not defined	33
bef	fore MPI-3.0. (End of ra	tionale.)	34
			35
Evampl	o 18 Using MPL GET	_ADDRESS for an array.	36
ылатр	C 4.0 Comg WI I_OLI.	$_$	37
REAL	A(100,100)		38
	GER(KIND=MPI_ADDRESS	_KIND) I1, I2, DIFF	39
CALL	MPI_GET_ADDRESS(A(1	,1), I1, IERROR)	40
CALL	MPI_GET_ADDRESS(A(1	0,10), I2, IERROR)	41
DIFF	= I2 - I1		42
		sizeofreal; the values of I1 and I2 are	43
! imple	mentation dependent.		44
			45

Advice to users.C users may be tempted to avoid the usage of46MPI_GET_ADDRESS and rely on the availability of the address operator &. Note,47however, that & cast-expression is a pointer, not an address. ISO C does not require48

Unofficial Draft for Comment Only

	1		-	nter (or the pointer cast to int) be the absolute address of the	
	2	object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such			
	$\frac{3}{4}$				
	5	machines as well. (<i>End of advice to users.</i>)			
	6				
	7	Advice to users. To prevent problems with the argument copying and register opti-			
ticket238-J.	8	mization done by Fortran compilers, please note the hints in subsections "Problems			
	9	Due to Data Copying and Sequence Association," and "A Problem with Register			
ticket238-J.	10	Optimization" in Section $16.2.10$ on pages 673 and 679 .]Sections $16.2.10-16.2.20$,			
ticket236-H.	11	especially in Sections 16.2.12 and 16.2.13 on pages 673-676 about "Problems Du			
	12			ence Association with Subscript Triplets" and "Vector Subscripts",	
ticket238-J.		 and in Sections 16.2.16 to 16.2.19 on pages 679 to 688 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (End of advice to users.) The following auxiliary function provides useful information on derived datatypes. 			
	14				
	15				
	16 17				
	18				
	19 20	MPI_TYPE	_SIZE(datatype, si	size)	
	20	IN	datatype	datatype (handle)	
	22	OUT	size	datatype size (integer)	
	23				
	24	<pre>int MPI_Type_size(MPI_Datatype datatype, int *size)</pre>			
ticket-248T.		MPI_Type_size(datatype, size, ierror) BIND(C)			
	26				
	27 28	TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: size			
	29	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
	30				
	31	MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR			
	32				
	33	$\{ \texttt{int MPI} :$:Datatype::Get_s	_size() const(binding deprecated, see Section 15.2) }	
	34	MPI_T	YPE_SIZE returns	ns the total size, in bytes, of the entries in the type signature	
	35	associated with datatype; i.e., the total size of the data in a message that would be created			
	36 27	with this d	latatype. Entries	that occur multiple times in the datatype are counted with	
ticket 265.	37 	their multi	plicity.		
	39			e OUT parameter cannot express the value to be returned (e.g.,	
	40	if the para	meter is too small	l to hold the output value), it is set to MPI_UNDEFINED.	
	41				
	42	4.1.6 Lov	ver-Bound and Up	pper-Bound Markers	
	43			ne explicitly the lower bound and upper bound of a type map,	
	44	and override the definition given on page 113. This allows one to define a datatype that has			
	45	"holes" at its beginning or its end, or a datatype with entries that extend above the upper			
	46	bound or below the lower bound. Examples of such usage are provided in Section 4.1.14.			
	47 48	Also, the user may want to overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide default alignment			
		bounds and	u extents. E.g., a	$\iota \cup$ computer may above the user to overlide default alignment	

rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional "pseudo-datatypes," MPI_LB and MPI_UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space $(extent(MPI_LB) = extent(MPI_UB) = 0)$. They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 4.9 Let D = (-3, 0, 6); $T = (MPI_LB, MPI_INT, MPI_UB)$, and B = (1, 1, 1). Then a call to MPI_TYPE_STRUCT(3, B, D, T, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence {(lb, -3), (int, 0), (ub, 6)}. If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the sequence {(lb, -3), (int, 0), (int,9), (ub, 15)}. (An entry of type ub can be deleted if there is another entry of type ub with a higher displacement; an entry of type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\$$

then the **lower bound** of *Typemap* is defined to be

$$lb(Typemap) = \begin{cases} \min_{j} disp_{j} & \text{if no entry has basic type Ib} \\ \min_{j} \{ disp_{j} \text{ such that } type_{j} = \mathsf{Ib} \} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of *Typemap* is defined to be

$$ub(Typemap) = \begin{cases} \max_{j}(disp_{j} + sizeof(type_{j})) + \epsilon & \text{if no entry has basic type ub} \\ \max_{j}\{disp_{j} \text{ such that } type_{j} = ub\} & \text{otherwise} \end{cases}$$

Then

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least non-negative increment needed to round extent(Typemap) to the next multiple of $\max_i k_i$. In Fortran, whether the alignments k_i are computed according to the alignments used by the compiler in common blocks, SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE nor BIND(C), is implementation-dependent.

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

Rationale. Before Fortran 2003, MPI_TYPE_CREATE_STRUCT could be applied to Fortran common blocks and SEQUENCE derived types. With Fortran 2003, this list was extended by BIND(C) derived types and MPI implementors have implemented the alignments k_i differently, i.e., some based on the alignments used in SEQUENCE derived types, and others according to BIND(C) derived types. (End of rationale.)

Advice to implementors. In Fortran, it is generally recommended to use BIND(C) derived types instead of common blocks or SEQUENCE derived types. Therefore it is recommended to calculate the alignments k_i based on BIND(C) derived types. (End of advice to implementors.)

Unofficial Draft for Comment Only

³³
³⁴ ticket229.2.
³⁵
³⁶
³⁷
³⁸ ticket229.2.
³⁹
⁴⁰
⁴¹

1

2

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18 19 20

29 30

31

32

42

43

44

45 46

47

1	
2	
3	Advice to users. Structures combining different basic datatypes should be defined
4	so that there will be no gaps based on alignment rules, and if used as an array of
5	structures, then also without such an alignment-gap at the end of the structure. In
6	MPI communication, the content of such gaps would not be communicated into the
7	receiver's buffer. For example, such an alignment-gap may occur between an odd
8	number of floats or REALs before a double or DOUBLE PRECISION data. Such gaps
9	may be added explicitly to both the structure and the MPI derived datatype handle
10	because the communication of a contiguous derived datatype may be significantly
11	faster than the communication of one that is non-contiguous due to such alignment-
12	gaps.
13	Example: Instead of
14	
15	TYPE $DIND(C)$ my data
16	TYPE, BIND(C) :: my_data REAL, DIMENSION(3) :: x
17	
18	! there may be a gap of the size of one REAL
19	! if the alignment of a DOUBLE PRECISION is
20	! two times the size of a REAL
21	DOUBLE PRECISION: p
22	END TYPE
23	
24	one should define
25	
26	TYPE, BIND(C) :: my_data
27	REAL, DIMENSION(3) :: x
28	REAL :: gap1
29	DOUBLE PRECISION: p
30	END TYPE
31	
32	and also including gap1 in the matching MPI derived datatype. It is required that all
33	processes in a communication add the same gaps, i.e., defined with the same basic
34	datatype. Both, the original and the modified structures are portable, but may have
35	different performance implications for the communication and memory accesses during
36	computation on systems with different alignment values.
37	
38	In principle, a compiler may define an additional alignment rule for structures, e.g., to
39	use at least 4 or 8 byte alignment although the content may have a max_ik_i alignment
40	less than this structure alignment. To keep an application portable, it is therefore
41	recommended to always resize structure derived datatype handles if used in an array
41	of structures, see the Example in Section 16.2.15 on page 677. (End of advice to
42	users.)
43 44	
44	
46	4.1.7 Extent and Bounds of Datatypes

The following function replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT [. It also returns] and also return address and count sized integers,

ticket265. 48 ticket265. ticket265.

46

respectively, in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

IN	datatype	datatype to get information on (handle)	6
OUT	lb	lower bound of datatype (integer)	7
OUT	extent	extent of datatype (integer)	8
001	externe	extent of datatype (meger)	9 10
int MPI T	vpe get extent(MPI Dataty	pe datatype, MPI_Aint *1b,	11
	MPI_Aint *extent)	F	12
MDT Trme	met eutent(detetune]h	ertent ierner) RIND(C)	$_{\scriptscriptstyle 13}$ ticket-248T.
	get_extent(datatype, lb, MPI_Datatype), INTENT(IN)		14
	• •	, INTENT(OUT) :: 1b, extent	15
	ER, OPTIONAL, INTENT(OUT)		16
			17
	GET_EXTENT(DATATYPE, LB,	EXTENT, IERROR)	18 19
	ER DATATYPE, IERROR ER(KIND = MPI_ADDRESS_KIN		20
TNIEG	ER(KIND - FILL_RDDRESS_KIN		21
$\{void MPI$	• -	PI::Aint& lb, MPI::Aint& extent)	22
	const(binding deprecated	d, see Section 15.2) }	23
Return	ns the lower bound and the e	extent of datatype (as defined in Section $4.1.6$ on	24

 25 ticket 265.

26

27

page 112).For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

28 MPI allows one to change the extent of a datatype, using lower bound and upper 29bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of 30 successive datatypes that are replicated by datatype constructors, or are replicated by the 31count argument in a send or receive call. However, the current mechanism for achieving it is painful; also it is restrictive. MPI_LB and MPI_UB are "sticky": once present in a 32 33 datatype, they cannot be overridden (e.g., the upper bound can be moved up, by adding 34a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes. The use of MPI_LB and MPI_UB 3536 is deprecated. 37

			38		
		D(oldtype, lb, extent, newtype)			
	E_CREATE_RESIZEL	(oldrype, ib, extent, newrype)	39		
IN	oldtype	input datatype (handle)	40		
IN	lb	new lower bound of datatype (integer)	41		
IIN	U		42		
IN	extent	new extent of datatype (integer)	43		
OUT	newtype	output datatype (handle)	44		
			45		
+ MPT	ad(MPT Datature oldture MPT Aint lb MPT Aint	46			
<pre>int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint</pre>					
	ercent, mr_	pararihe when ribes			

 48 ticket-248T.

Unofficial Draft for Comment Only

1 2

3 4

	1 2 3 4 5	INTEGE TYPE(M TYPE(M) :: newtype	
	6 7 8 9	INTEGE	CREATE_RESIZED(OLDTYPE, L ER OLDTYPE, NEWTYPE, IERR ER(KIND=MPI_ADDRESS_KIND)		
	10 11	{MPI::Data		<pre>te_resized(const MPI::Aint lb, nt) const(binding deprecated, see Section 15.2) }</pre>	
Returns in newtype a handle to a new datatype that is identical to oldtype, the lower bound of this new datatype is set to be lb, and its upper bound is a + extent. Any previous lb and ub markers are erased, and a new pair of lower upper bound markers are put in the positions indicated by the lb and extent This affects the behavior of the datatype when used in communication opera count > 1, and when used in the construction of new derived datatypes.					
20				ommended that users use these two new functions, s to set and access lower bound, upper bound and e to users.)	
	23 24	4.1.8 True	e Extent of Datatypes		
	25 26 27 28 29 30 31 32	mented on the process, one nodes. How that needs t	top of point-to-point routines e will need to allocate some to ever, the datatype extent can to be allocated, if the user has	Section 5.5 on page 160) as a spanning tree imple- . Since the receive buffer is only valid on the root emporary space for receiving data on intermediate not be used as an estimate of the amount of space modified the extent using the MPI_UB and MPI_LB urns the true extent of the datatype.	
	33	MPI_TYPE	_GET_TRUE_EXTENT(dataty	<pre>rpe, true_lb, true_extent)</pre>	
	34 35	IN	datatype	datatype to get information on (handle)	
	36	OUT	true_lb	true lower bound of datatype (integer)	
	37	OUT	true_extent	true size of datatype (integer)	
ticket-248T.	42	<pre>int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,</pre>			
	43 44 45	INTEGE		, INTENT(OUT) :: true_lb, true_extent	
	46 47 48	INTEGE	ET_TRUE_EXTENT(DATATYPE, ER DATATYPE, IERROR ER(KIND = MPI_ADDRESS_KIN	TRUE_LB, TRUE_EXTENT, IERROR) D) TRUE_LB, TRUE_EXTENT	

1 {void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb, $\mathbf{2}$ MPI::Aint& true_extent) const(binding deprecated, see Section 15.2) } 3 true_lb returns the offset of the lowest unit of store which is addressed by the datatype, 4 i.e., the lower bound of the corresponding typemap, ignoring MPI_LB markers. true_extent 5 returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring 6 MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap 7 associated with datatype is 8 9 $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$ 10 Then 11 12 $true_lb(Typemap) = min_i \{ disp_i : type_i \neq \mathbf{lb}, \mathbf{ub} \},\$ 13 $true_ub(Typemap) = max_i \{ disp_i + sizeof(type_i) : type_i \neq \mathbf{lb}, \mathbf{ub} \},\$ 1415and 1617 $true_extent(Typemap) = true_ub(Typemap) - true_lb(typemap).$ 18 (Readers should compare this with the definitions in Section 4.1.6 on page 112 and Sec-19 20tion 4.1.7 on page 114, which describe the function MPI_TYPE_GET_EXTENT.) 21The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed. 22 ticket 265. For both functions, if either OUT parameter cannot express the value to be returned 23(e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED. 24 2526Commit and Free 4.1.9 27A datatype object has to be **committed** before it can be used in a communication. As 28 an argument in datatype constructors, uncommitted and also committed datatypes can be 29 used. There is no need to commit basic datatypes. They are "pre-committed." 30 3132 MPI_TYPE_COMMIT(datatype) 33 INOUT datatype datatype that is committed (handle) 34 35 int MPI_Type_commit(MPI_Datatype *datatype) 36 ₃₇ ticket-248T. MPI_Type_commit(datatype, ierror) BIND(C) 38 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI_TYPE_COMMIT(DATATYPE, IERROR) 42INTEGER DATATYPE, IERROR 43 {void MPI::Datatype::Commit() (binding deprecated, see Section 15.2) } 44 The commit operation commits the datatype, that is, the formal description of a com-4546

munication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

1 2 3 4 5	representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (<i>End of advice to implementors.</i>)
6 7	
8	Example 4.10 The following gode fragment gives examples of using MPL TYPE COMMIT
10	INTEGER type1, type2
12	
13	
14	! now type1 can be used for communication
15	cypez – cypei
16	i typez can be used for communication
17	! (It is a handle to same object as type)
19	CALL MPI_ITPE_VECTOR(3, 5, 4, MPI_REAL, type1, Terr)
20	! new uncommitted type object created
21	CALL MFI_ITFE_COMMIT(type1, IEII)
22	! now type: can be used anew for communication
23	
24	
25	MPI_TYPE_FREE(datatype)
26 27	
28 ticket-248T. 29	int MPI_Type_free(MPI_Datatype *datatype)
	MPI Type free(datatype, jerror) BIND(C)
30	TYPE(MPI Datatype), INTENT(INOUT) :: datatype
32	INTEGER OPTIONAL INTENT(OUT) ··· ierror
33	
34	
35	
36	
37	marks the datatype object associated with datatype for dealocation and sets datatype
38	to with DATATILE_NOLE. Any communication that is currently using this datatype with
39 40	complete normany. Treening a datatype does not ancer any other datatype that was built
41	from the need datatype. The system behaves as it input datatype arguments to derived
42	datatype constitucions are passed by value.
43	Advice to implementors. The implementation may keep a reference count of active
44	
45	
46	datatype arguments, rather then copying them. In this case, one needs to keep track
47	$\mathbf{J}_{\mathbf{I}}$
48	be freed. (End of advice to implementors.)

47

48

```
4.1.10 Duplicating a Datatype
                                                                                             1
                                                                                             \mathbf{2}
                                                                                             3
                                                                                             4
MPI_TYPE_DUP(oldtype, newtype)
                                                                                               ticket252-W.
                                                                                             5
  IN
            [ticket252-W.]oldtype
                                         datatype (handle)
  OUT
                                                                                             7 ticket252-W.
            newtype
                                         copy of oldtype (handle)
                                                                                             9
int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
                                                                                              ticket252-W
                                                                                             ^{10} ticket-248T.
MPI_Type_dup(oldtype, newtype, ierror) BIND(C)
                                                                                             11
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                            12
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                             13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            14
                                                                                            ^{15} ticket252-W.
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
                                                                                            ^{16} ticket252-W.
    INTEGER OLDTYPE, NEWTYPE, IERROR
                                                                                             17
{MPI::Datatype MPI::Datatype::Dup() const(binding deprecated, see Section 15.2) }
                                                                                            18
                                                                                            19
    MPI_TYPE_DUP is a type constructor which duplicates the existing
                                                                                            20
type with associated key values. For each key value, the respective copy callback function
                                                                                            21
determines the attribute value associated with this key in the new communicator; one
                                                                                            22
particular action that a copy callback may take is to delete the attribute from the new
                                                                                            23
                                                                                               ticket252-W
datatype. Returns in newtype a new datatype with exactly the same properties as oldtype
                                                                                            24
and any copied cached information, see Section 6.7.4 on page 288. The new datatype has
                                                                                             25
identical upper bound and lower bound and yields the same net result when fully decoded
                                                                                             26
with the functions in Section 4.1.13. The newtype has the same committed state as the old
                                                                                            27
                                                                                               ticket252-W.
oldtype.
                                                                                             28
                                                                                            29
       Use of General Datatypes in Communication
4.1.11
                                                                                            30
Handles to derived datatypes can be passed to a communication call wherever a datatype
                                                                                            ^{31}
argument is required. A call of the form MPI_SEND(buf, count, datatype, ...), where
                                                                                            32
count > 1, is interpreted as if the call was passed a new datatype which is the concatenation
                                                                                            33
of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is
                                                                                            34
equivalent to,
                                                                                             35
                                                                                            36
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
                                                                                            37
MPI_TYPE_COMMIT(newtype)
                                                                                            38
MPI_SEND(buf, 1, newtype, dest, tag, comm).
                                                                                             39
                                                                                             40
Similar statements apply to all other communication functions that have a count and
                                                                                             41
datatype argument.
                                                                                            42
    Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is
                                                                                             43
executed, where datatype has type map,
                                                                                             44
     \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\
                                                                                             45
                                                                                             46
and extent extent. (Empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in
```

Unofficial Draft for Comment Only

the type map, but they affect the value of *extent*.) The send operation sends $n \cdot count$

¹ entries, where entry $i \cdot n + j$ is at location $addr_{i,j} = buf + extent \cdot i + disp_j$ and has type ² $type_j$, for i = 0, ..., count - 1 and j = 0, ..., n - 1. These entries need not be contiguous, nor ³ distinct; their order can be arbitrary.

⁴ The variable stored at address $addr_{i,j}$ in the calling program should be of a type that ⁵ matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains ⁶ $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ has type $type_j$.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

```
\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\
```

with extent *extent*. (Again, empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) This receive operation receives $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ is at location buf $+ extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot \text{count}$; the $i \cdot n + j$ -th element of the message should have a type that matches $type_j$.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

Example 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
^{24}
     . . .
     CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
25
     CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
26
     CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
27
     . . .
28
     CALL MPI_SEND( a, 4, MPI_REAL, ...)
29
     CALL MPI_SEND( a, 2, type2, ...)
30
     CALL MPI_SEND( a, 1, type22, ...)
^{31}
     CALL MPI_SEND( a, 1, type4, ...)
32
33
     . . .
     CALL MPI_RECV( a, 4, MPI_REAL, ...)
34
     CALL MPI_RECV( a, 2, type2, ...)
35
     CALL MPI_RECV( a, 1, type22, ...)
36
     CALL MPI_RECV( a, 1, type4, ...)
37
38
```

Each of the sends matches any of the receives.

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

 $45 \\ 46$

 $\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}.$

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where

Unofficial Draft for Comment Only

10

21

22

ticket265. ticket265.

$0 \le k \le \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query [function]functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.				
uno quorj			3	
			4	
MPI_GET	5			
IN	status	return status of receive operation (Status)	6	
IN	datatype	datatype used by receive operation (handle)	7	
		·- · · · · · · · · · · · · · · · · · ·	8	
OUT	count	number of received basic elements (integer)	9	
	_		10	
int MPI_		MPI_Status *status, MPI_Datatype datatype,	11 ticket 140.	
	int *count)		$^{12}_{13}$ ticket-248T.	
MPI_Get_	elements(status, d	latatype, count, ierror) BIND(C)		
TYPE	C(MPI_Status), INTE	NT(IN) :: status	14 15	
TYPE	(MPI_Datatype), IN	ITENT(IN) :: datatype	16	
INTE	GER, INTENT(OUT) :	: count	17	
INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror	18	
MPT GFT	FI FMFNTS (STATUS D	DATATYPE, COUNT, IERROR)	19	
	· · · · ·	TUS_SIZE), DATATYPE, COUNT, IERROR	20	
			21	
{int MPI		<pre>nents(const MPI::Datatype& datatype) const(binding</pre>	22	
	deprecated, see	Section 15.2 }	23 ticket 265.	
			24	
			25	
MPI_GET	_ELEMENTS_X(stat	us, datatype, count)	26	
IN	status	return status of receive operation (Status)	27	
IN	datatype	datatype used by receive operation (handle)	28 29	
OUT	count	number of received basic elements (integer)	30	
			31	
int MPT	Get elements x(MPI		32	
1110 111 1_	MPI_Count *co	·· · · · ·	33	
	_		$_{34}$ ticket-248T.	
		datatype, count, ierror) BIND(C)	35	
	(MPI_Status), INTE		36	
		ITENT(IN) :: datatype	37	
		NT_KIND), INTENT(OUT) :: count	38	
	GER, UPIIUNAL, INI	ENT(OUT) :: ierror	39	
MPI_GET_	ELEMENTS_X(STATUS,	DATATYPE, COUNT, IERROR)	40	
INTE	GER STATUS(MPI_STA	TUS_SIZE), DATATYPE, IERROR	41	
INTE	GER (KIND=MPI_COUN	IT_KIND) COUNT	42	
The	datatyne argument ch	ould match the argument provided by the receive call that	43	
		h functions, if the OUT parameter cannot express the value	44	
		rameter is too small to hold the output value), it is set to	45	
MPI_UND			46	

The previously defined function MPI_GET_COUNT[,] (Section 3.2.5), has a different

$_{48}^{48}$ ticket 265.

```
1
               behavior. It returns the number of "top-level entries" received, i.e. the number of "copies"
         \mathbf{2}
               of type datatype. In the previous example, MPI_GET_COUNT may return any integer value
         3
               k, where 0 < k < \text{count.} If MPI_GET_COUNT returns k, then the number of basic ele-
         4
               ments received (and the value returned by MPI_GET_ELEMENTS) is n \cdot k. If the number
         \mathbf{5}
               of basic elements received is not a multiple of n, that is, if the receive operation has not
          6
               received an integral number of datatype "copies," then MPI_GET_COUNT returns the value
ticket265.<sup>7</sup>
               MPI_UNDEFINED. The datatype argument should match the argument provided by the re-
          8
               ceive call that set the status variable.]
         9
               Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.
         10
         11
               . . .
         12
               CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
         13
               CALL MPI_TYPE_COMMIT(Type2, ierr)
         14
               . . .
         15
               CALL MPI_COMM_RANK(comm, rank, ierr)
         16
               IF (rank.EQ.0) THEN
         17
                      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
         18
                      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
         19
               ELSE IF (rank.EQ.1) THEN
         20
                      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
         21
                      CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                                           ! returns i=1
         22
                      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                                          ! returns i=2
         23
                      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
         24
                      CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                                           ! returns i=MPI_UNDEFINED
         25
                      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                                          ! returns i=3
         26
               END IF
         27
         28
                   The function MPI_GET_ELEMENTS can also be used after a probe to find the number
ticket 265. ^{29}
               of elements in the probed message. Note that the [two] functions MPI_GET_COUNT [and],
ticket265.<sup>30</sup>
               MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return the same values when they
ticket265.<sup>31</sup>
               are used with basic datatypes so long as the limits of their respective count arguments are
ticket265. 32
               not exceeded.
         33
         34
                     Rationale. The extension given to the definition of MPI_GET_COUNT seems natural:
         35
                    one would expect this function to return the value of the count argument, when the
         36
                    receive buffer is filled. Sometimes datatype represents a basic unit of data one wants
         37
                    to transfer, for example, a record in an array of records (structures). One should be
         38
                    able to find out how many components were received without bothering to divide by
         39
                    the number of elements in each component. However, on other occasions, datatype
         40
                    is used to define a complex layout of data in the receiver memory, and does not
         41
                    represent a basic unit of data for transfers. In such cases, one needs to use the
         42
                     function MPI_GET_ELEMENTS. (End of rationale.)
         43
                     Advice to implementors.
                                                The definition implies that a receive cannot change the
         44
                     value of storage outside the entries defined to compose the communication buffer. In
         45
                    particular, the definition implies that padding space in a structure should not be mod-
         46
         47
                    ified when such a structure is copied from one process to another. This would prevent
                     the obvious optimization of copying the structure, together with the padding, as one
         48
```

contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (*End of advice to implementors.*)

4.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

Variables belong to the same **sequential storage** if they belong to the same array, to the same **COMMON** block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

- 1. The function MPI_GET_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
- 2. The **buf** argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
- 3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.
- 4. If v is a valid address then MPI_BOTTOM + v is a valid address.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully executed on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

Advice to users. It is not expected that MPI implementations will be able to detect erroneous, "out of bound" displacements — unless those overflow the user address space — since the MPI call may not know the extent of the arrays and records in the host program. (*End of advice to users.*)

Advice to implementors. There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (*End of advice to implementors.*)

Unofficial Draft for Comment Only

 $\overline{7}$

 31

	1	4.1.13	Decoding a Datatype					
	2 3 4	are seve	eral cases where accessing	to specify an arbitrary layout of data in memory. There the layout information in opaque datatype objects would				
	5			be be be be a number of uses outside MPI. Further- isplay internal information about a datatype. To achieve				
	6			are provided. The two functions in this section are used				
	1			recreate the calling sequence used in their initial defini-				
	9			a user to determine the type map and type signature of a				
	10	datatype.						
	11							
	12 13	MPI_TY	YPE_GET_ENVELOPE(dat biner)	atype, num_integers, num_addresses, num_datatypes, com-				
	14 15	IN	datatype	datatype to access (handle)				
	16 17	OUT	num_integers	number of input integers used in the call constructing combiner (non-negative integer)				
	18 19	OUT	num_addresses	number of input addresses used in the call construct- ing combiner (non-negative integer)				
	20 21	OUT	num_datatypes	number of input datatypes used in the call construct-				
	22			ing combiner (non-negative integer)				
	23	OUT	combiner	combiner (state)				
	24							
ticket-248T.	25 26	<pre>int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>						
UICKEU-2401.		MPI_Typ	pe_get_envelope(dataty	pe, num_integers, num_addresses, num_datatypes,				
	29		combiner, ierro					
	30	TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,						
	31	combiner						
	32 33		TEGER, OPTIONAL, INTEN	T(OUT) :: ierror				
		MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,						
	35	COMBINER, IERROR)						
	36	INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,						
	37 38	IEF	RROR					
	39	{void M	MPI::Datatype::Get_env	elope(int& num_integers, int& num_addresses,				
	40			rpes, int& combiner) const(binding deprecated, see				
	41		Section 15.2 }					
	42	For	the given datatype, MPI_	TYPE_GET_ENVELOPE returns information on the num-				
	43 44			sed in the call that created the datatype. The number-of-				
		0		used to provide sufficiently large arrays in the decoding NTS. This call and the meaning of the returned values is				
	46			flects the MPI datatype constructor call that was used in				
	47		g datatype.					
48								

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from MPI_TYPE_GET_CONTENTS may be used with either to produce the same outcome. C calls MPI_Type_hindexed and MPI_Type_create_hindexed are always effectively the same while the Fortran call MPI_TYPE_HINDEXED will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (*End of rationale.*)

The list below has the values that can be returned in **combiner** on the left and the call associated with them on the right.

MPI_COMBINER_NAMED	a named predefined datatype	20
MPI_COMBINER_DUP	MPI_TYPE_DUP	21
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	22
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	23
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR from Fortran	24
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR from C or C++	25
	and in some case Fortran	26
	or MPI_TYPE_CREATE_HVECTOR	27
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	28
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED from Fortran	29
MPI_COMBINER_HINDEXED	$MPI_TYPE_HINDEXED \text{ from } \mathrm{C} \text{ or } \mathrm{C}{++}$	30
	and in some case Fortran	31
	or MPI_TYPE_CREATE_HINDEXED	32
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	33
[ticket280.]MPI_COMBINER_HINDEXED_E		34
	MPI_TYPE_CREATE_HINDEXED_BLOCK	35
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT from Fortran	36
MPI_COMBINER_STRUCT	$MPI_TYPE_STRUCT \text{ from } \mathrm{C} \text{ or } \mathrm{C}{+}{+}$	37
	and in some case Fortran	38
	or MPI_TYPE_CREATE_STRUCT	39
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	40
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	41
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	42
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX	43
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	44
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	45
		40 46
		47
Table 4.1: combiner values return	rned from MPI_TYPE_GET_ENVELOPE	48

 $\mathbf{2}$

 $\mathbf{5}$

	MED then datatype is a named predefined datatype. arguments, we sometimes need to differentiate whether size argument. For example, there are two combin- CTOR_INTEGER and MPI_COMBINER_HVECTOR. The ll from Fortran, and the latter is used if it was the c, on systems where MPI_ADDRESS_KIND = arguments and address size arguments are the same), OR may be returned for a datatype constructed by a Fortran. Similarly, MPI_COMBINER_HINDEXED may d by a call to MPI_TYPE_HINDEXED from Fortran, e returned for a datatype constructed by a call to On such systems, one need not differentiate construc- from constructors that take integer arguments, since ls all use address sized arguments so two combiners				
1 1 2 2 2	17 18 19 20 21 22	tion m could than t	hay have been truncated. ' be subject to truncation i he size of an address. (En	iginal call, it is important to know if address informa- The deprecated calls from Fortran for a few routines in the case where the default INTEGER size is smaller d of rationale.) creation call for a datatype can be obtained from the	
2	24 25 26 27	MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, ray_of_integers, array_of_addresses, array_of_datatypes)			
	28	IN	datatype	datatype to access (handle)	
3	29 30	IN	max_integers	number of elements in array_of_integers (non-negative integer)	
:	31 32 33	IN	max_addresses	number of elements in <code>array_of_addresses</code> (non-negative integer)	
	34 35	IN	max_datatypes	number of elements in <code>array_of_datatypes</code> (non-negative integer)	
3	36 37	OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)	
:	38 39 40	OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)	
4	41 42 43	OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)	
4	45 46 47	nt MPI_Ty			

<pre>MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,</pre>	1
<pre>array_of_integers, array_of_addresses, array_of_datatypes,</pre>	2
ierror) BIND(C)	3
TYPE(MPI_Datatype), INTENT(IN) :: datatype	4
<pre>INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes</pre>	5
<pre>INTEGER, INTENT(OUT) :: array_of_integers(max_integers)</pre>	6
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::	7
array_of_addresses(max_addresses)	8
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	12
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	13
IERROR)	14
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES, ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	15
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)	16
INIEGER(KIND=MPI_ADDRESS_KIND) ARRAY_UF_ADDRESSES(*)	17
<pre>{void MPI::Datatype::Get_contents(int max_integers, int max_addresses,</pre>	18
<pre>int max_datatypes, int array_of_integers[],</pre>	19
<pre>MPI::Aint array_of_addresses[],</pre>	20
<pre>MPI::Datatype array_of_datatypes[]) const(binding deprecated, see</pre>	21
Section 15.2 }	22
determs must be a predefined unperced on a derived determore the call is amongous if	23
datatype must be a predefined unnamed or a derived datatype; the call is erroneous if datatype is a predefined named datatype.	24
	25
The values given for max_integers, max_addresses, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, and num_datatypes, respectively,	26
in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.	27
In the can MFI_TTFE_GET_ENVELOPE for the same datatype argument.	28
<i>Rationale.</i> The arguments max_integers, max_addresses, and max_datatypes allow for	29
error checking in the call. (End of rationale.)	30
· · · · · · · · · · · · · · · · · · ·	31

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale.The definition of datatype equivalence implies that equivalent predefined 45 datatypes are equal.By requiring the same handle for named predefined datatypes, 46 it is possible to use the == or .EQ. comparison operator to determine the datatype 47 involved. (End of rationale.) 48

Unofficial Draft for Comment Only

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (End of advice to implementors.)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type INTEGER. In the preferred calls, the address arguments are of type

INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

- Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (End of rationale.)
- 27 28

29

30

 31

32

21

22

23

24

25

26

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:

³³ PARAMETER (LARGE = 1000)

34 INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 35INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE) 36 i CONSTRUCT DATATYPE TYPE (NOT SHOWN) 37 CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR) 38 IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN 39 WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, & 40 " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE 41 CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR) 42ENDIF 43 CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR) 4445or in C the analogous calls of: 4647#define LARGE 1000 48int ni, na, nd, combiner, i[LARGE];

1

2

3

4

5

6

7

8

9

10

11

MPI_Aint a[LARGE];							
MPI_Datatype type, d			2				
/* construct datatype	e type (not shown)	*/	3				
MPI_Type_get_envelope	e(type, ∋, &na, &	knd, &combiner);	4				
if ((ni > LARGE) ((na > LARGE) (no	1 > LARGE)) {	5				
fprintf(stderr, "ni	i, na, or nd = %d %	<pre>%d %d returned by ", ni,</pre>	na, nd); ⁶				
fprintf(stderr, "MF	PI_Type_get_envelop	be is larger than LARGE	= %d\n", ⁷				
LARGE);							
MPI_Abort(MPI_COMM_	_WORLD, 99);		9				
};			10				
MPI_Type_get_contents	s(type, ni, na, nd	, i, a, d);	11				
	•••		12				
The $C++$ code is in	analogy to the C cod	le above with the same value	es returned. ¹³				
In the descriptions t	that follow, the lower	case name of arguments is u	used. 14				
If combiner is MPI_	COMBINER_NAMED t	hen it is erroneous to call	15				
MPI_TYPE_GET_CONT	ENTS.		16				
If combiner is MPI_C	COMBINER_DUP then		17				
		Eastern la satism	18				
Constructor argument	C & C++ location	Fortran location	19				
oldtype	d[0]	D(1)	20				
and $ni = 0$, $na = 0$, $nd =$			21				
If combiner is MPI_C	COMBINER_CONTIGUC	OUS then	22				
Constructor argument	C & C++ location	Fortran location	23				
count	i[0]	I(1)	24				
oldtype	d[0]	D(1)	25				
and $ni = 1$, $na = 0$, $nd =$	= 1.		26				
, , ,	COMBINER_VECTOR tl	nen	27				
Constructor argument	C & C++ location	Fortran location	28 29				
count	i[0]	I(1)	30				
blocklength	i[1]	I(2)					
stride	i[2]	I(3)	31				
oldtype	d[0]	D(1)	32				
			33				
and $ni = 3$, $na = 0$, $nd =$		INTEGER or MPI_COMBINER	³⁴ R HVFCTOR then ³⁵				
			36 36				
Constructor argument	C & C++ location	Fortran location	37				
count	i[0]	I(1)	38				
blocklength	$\mathrm{i}[1]$	I(2)	39				
stride	a[0]	$\mathrm{A}(1)$	40				
oldtype	d[0]	D(1)					
and $ni = 2$, $na = 1$, $nd =$	= 1.		41				
	COMBINER_INDEXED t	hen	42 43				
Constructor argument	C & C++ location	Fortran location	- 44				
count	$\frac{C \& C + 10 \text{ location}}{i[0]}$	I(1)	- 45				
array_of_blocklengths	i[1] to i[i[0]]	I(1) I(2) to I(I(1)+1)	46				
			40				
array_of_displacements			48				
oldtype	d[0]	D(1)	- 40				

1 and $ni = 2^*count+1$, na = 0, nd = 1. $\mathbf{2}$ If combiner is MPI_COMBINER_HINDEXED_INTEGER or MPI_COMBINER_HINDEXED then 3 $\overline{C} \& \overline{C} + + \text{location}$ Constructor argument Fortran location 4 i[0] count I(1)5array_of_blocklengths i[1] to i[i[0]]I(2) to I(I(1)+1)6 a[0] to a[i[0]-1]A(1) to A(I(1))array_of_displacements 7 D(1)oldtype d[0]8 and ni = count+1, na = count, nd = 1. 9 If combiner is MPI_COMBINER_INDEXED_BLOCK then 10 11 C & C++ locationConstructor argument Fortran location 12i[0] I(1)count 13 i[1] I(2)blocklength 14array_of_displacements i[2] to i[i[0]+1]I(3) to I(I(1)+2)15D(1)oldtype d[0]16ticket280. and ni = count+2, na = 0, nd = 1. 17 If combiner is MPI_COMBINER_HINDEXED_BLOCK then 18 Constructor argument C & C++ location Fortran location 19 count i[0] I(1)20i[1] blocklength I(2)21array_of_displacements a[0] to a[i[0]-1]A(1) to A(I(1))22 oldtype d[0]D(1)23 24 and ni = 2, na = count, nd = 1. 25If combiner is MPI_COMBINER_STRUCT_INTEGER or MPI_COMBINER_STRUCT then 26C & C++ location Constructor argument Fortran location 27i[0] count I(1)28array_of_blocklengths i[1] to i[i[0]] I(2) to I(I(1)+1)29 array_of_displacements a[0] to a[i[0]-1]A(1) to A(I(1))30 D(1) to D(I(1))array_of_types d[0] to d[i[0]-1] 31 and ni = count+1, na = count, nd = count. 32 If combiner is MPI_COMBINER_SUBARRAY then 33 34C & C++location Fortran location Constructor argument 35 ndims i[0]I(1)36 array_of_sizes i[1] to i[i[0]] I(2) to I(I(1)+1)37 array_of_subsizes i[i[0]+1] to i[2*i[0]]I(I(1)+2) to I(2*I(1)+1)38 array_of_starts i[2*i[0]+1] to i[3*i[0]]I(2*I(1)+2) to I(3*I(1)+1)39 order i[3*i[0]+1]I(3*I(1)+2]40 oldtype d[0]D(1)41 and ni = 3*ndims+2, na = 0, nd = 1. 42If combiner is MPI_COMBINER_DARRAY then 43 44 4546 4748

Constructor argument	C & C++ locatio	n Fortrar	n location	1
size	i[0]	l	$\overline{(1)}$	2
rank	i[1]	1	(2)	3
ndims	i[2]	I	(3)	4
array_of_gsizes	i[3] to $i[i[2]+2]$	I(4) to	I(I(3)+3)	5
array_of_distribs	i[i[2]+3] to $i[2*i[2]+3]$	-2] I(I(3)+4) t	o $I(2*I(3)+3)$	6
array_of_dargs	i[2*i[2]+3] to $i[3*i[2]$	+2] I(2*I(3)+4)	to $I(3*I(3)+3)$	7
$array_of_psizes$	i[3*i[2]+3] to $i[4*i[2]$	+2] I(3*I(3)+4)	to $I(4*I(3)+3)$	8
order	i[4*i[2]+3]	I(4*]	(3)+4)	9
oldtype	d[0]	Γ	0(1)	10
and $ni = 4*ndims+4$, na	= 0, nd = 1.			11
If combiner is MPI_C	COMBINER_F90_REAL	then		12 13
Constructor argument	C & C++ location	Fortran location	-	13
р	i[0]	I(1)		15
r	i[1]	I(2)		16
and $ni = 2$, $na = 0$, $nd =$	= 0.		-	17
	COMBINER_F90_COMP	LEX then		18
Constructor argument	C & C++ location	Fortran location	-	19
р	i[0]	I(1)	-	20
r	i[1]	I(2)		21 22
and $ni = 2$, $na = 0$, $nd =$	= 0.		-	22
	COMBINER_F90_INTEG	ER then		23 24
Constructor argument	C & C++ location	Fortran location	-	25
r	i[0]	I(1)	-	26
and $ni = 1$, $na = 0$, $nd =$	= 0.		-	27
If combiner is MPI_C	COMBINER_RESIZED t	hen		28 29
Constructor argument	C & C++ location	Fortran location	-	29 30
lb	a[0]	A(1)	-	31
extent	a[1]	A(2)		32
oldtype	d[0]	D(1)		33
and $ni = 0$, $na = 2$, $nd =$	= 1.		-	34
				35
4.1.14 Examples				36
The following examples i	llustrate the use of de	erived datatypes.		37
0 I		J 1		38 39
Example 4.13 Send an	d receive a section of	a 3D array.		40
DEAL - (100 100				41
REAL a(100,100,		alico director		42
	ce, twoslice, three MPT STATUS SIZE)	ESTICE, SIZEOILE	ar, myrank, ierr	43
INIEGER STATUS	(MPI_STATUS_SIZE)			44
C extract the se	ection a(1:17:2, 3	•11 2•10)		45
C extract the se C and store it i		.11, 2.10/		46
				47
				48

1 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr) $\mathbf{2}$ 3 CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr) 4 5С create datatype for a 1D section 6 CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr) 7 8 С create datatype for a 2D section 9 CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr) 1011С create datatype for the entire section CALL MPI_TYPE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice, 1213 threeslice, ierr) 1415CALL MPI_TYPE_COMMIT(threeslice, ierr) 16CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, 17 MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr) 18 19 **Example 4.14** Copy the (strictly) lower triangular part of a matrix. 2021REAL a(100,100), b(100,100) 22 INTEGER disp(100), blocklen(100), ltype, myrank, ierr 23INTEGER status(MPI_STATUS_SIZE) 2425С copy lower triangular part of array a 26С onto lower triangular part of array b 2728CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr) 29 30 С compute start and size of each column 31 DO i=1, 100 32 disp(i) = 100*(i-1) + i 33 blocklen(i) = 100-i 34 END DO 35 36 С create datatype for lower triangular part 37 CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr) 38 39 CALL MPI_TYPE_COMMIT(ltype, ierr) 40 CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, 41 ltype, myrank, 0, MPI_COMM_WORLD, status, ierr) 4243 **Example 4.15** Transpose a matrix. 4445 REAL a(100,100), b(100,100) 46 INTEGER row, xpose, sizeofreal, myrank, ierr 47 INTEGER status(MPI_STATUS_SIZE) 48

4.1. DERIVED DATATYPES

		1
С	transpose matrix a onto b	2
		3
	CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)	4 5
	CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)	6
	ORLE MIT_THE_EATENT(MIT_MERE, SIZEOTTEAT, TETT)	7
С	create datatype for one row	8
	CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)	9
		10
С	create datatype for matrix in row-major order	11
	CALL MPI_TYPE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)	12 13
	CALL MDI TYDE COMMIT(unage igno)	13 14
	CALL MPI_TYPE_COMMIT(xpose, ierr)	15
С	send matrix in row-major order and receive in column major order	16
-	CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100,	17
	MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)	18
		19
Exam	ple 4.16 Another approach to the transpose problem:	20
LAU		21
	REAL a(100,100), b(100,100)	22
	<pre>INTEGER disp(2), blocklen(2), type(2), row, row1, sizeofreal</pre>	23 24
	INTEGER myrank, ierr INTEGER status(MPI_STATUS_SIZE)	25
	INTEGER Status (MF1_STATOS_SIZE)	26
	CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)	27
	······································	28
С	transpose matrix a onto b	29
		30
	CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)	31
_		32
С	create datatype for one row	$33 \\ 34$
	CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)	35
С	create datatype for one row, with the extent of one real number	36
0	disp $(1) = 0$	37
	disp(2) = sizeofreal	38
	type(1) = row	39
	type(2) = MPI_UB	40
	blocklen(1) = 1	41
	blocklen(2) = 1	42
	CALL MPI_TYPE_STRUCT(2, blocklen, disp, type, row1, ierr)	43
	CALL MDI TYDE COMMIT(rout iorr)	44 45
	CALL MPI_TYPE_COMMIT(row1, ierr)	46
С	send 100 rows and receive in column major order	47
-	CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100,	48

```
1
                      MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
2
3
     Example 4.17 We manipulate an array of structures.
4
\mathbf{5}
     struct Partstruct
6
        {
\overline{7}
           int
                   class; /* particle class */
8
           double d[6]; /* particle coordinates */
9
           char
                   b[7]; /* some additional information */
10
        };
11
12
     struct Partstruct
                           particle[1000];
13
14
                   i, dest, rank, tag;
     int
15
     MPI_Comm
                   comm;
16
17
18
     /* build datatype describing structure */
19
20
     MPI_Datatype Particletype;
21
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
22
                   blocklen[3] = \{1, 6, 7\};
     int
23
     MPI_Aint
                   disp[3];
^{24}
     MPI_Aint
                  base;
25
26
27
     /* compute displacements of structure components */
28
29
     MPI_Address( particle, disp);
30
     MPI_Address( particle[0].d, disp+1);
^{31}
     MPI_Address( particle[0].b, disp+2);
32
     base = disp[0];
33
     for (i=0; i < 3; i++) disp[i] -= base;</pre>
34
35
     MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
36
37
        /* If compiler does padding in mysterious ways,
38
        the following may be safer */
39
40
     MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
41
                   blocklen1[4] = \{1, 6, 7, 1\};
     int
42
     MPI_Aint
                   disp1[4];
43
44
     /* compute displacements of structure components */
45
46
     MPI_Address( particle, disp1);
47
     MPI_Address( particle[0].d, disp1+1);
48
     MPI_Address( particle[0].b, disp1+2);
```

```
1
MPI_Address( particle+1, disp1+3);
                                                                                       \mathbf{2}
base = disp1[0];
                                                                                       3
for (i=0; i < 4; i++) disp1[i] -= base;</pre>
                                                                                       4
/* build datatype describing structure */
                                                                                       5
                                                                                       6
                                                                                       7
MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);
                                                                                       8
                                                                                       9
                                                                                       10
               /* 4.1:
                                                                                       11
         send the entire array */
                                                                                       12
MPI_Type_commit( &Particletype);
                                                                                       13
                                                                                       14
MPI_Send( particle, 1000, Particletype, dest, tag, comm);
                                                                                       15
                                                                                       16
                                                                                       17
               /* 4.2:
                                                                                       18
         send only the entries of class zero particles,
                                                                                       19
        preceded by the number of such entries */
                                                                                       20
                                                                                       21
MPI_Datatype Zparticles;
                             /* datatype describing all particles
                                 with class zero (needs to be recomputed
                                                                                       22
                                                                                       23
                                 if classes change) */
                                                                                       ^{24}
MPI_Datatype Ztype;
                                                                                       25
                                                                                       26
              zdisp[1000];
MPI_Aint
int
              zblock[1000], j, k;
                                                                                       27
              zzblock[2] = \{1,1\};
                                                                                       28
int
                                                                                       29
MPI_Aint
              zzdisp[2];
                                                                                       30
MPI_Datatype zztype[2];
                                                                                       31
                                                                                       32
/* compute displacements of class zero particles */
                                                                                       33
j = 0;
                                                                                       34
for(i=0; i < 1000; i++)</pre>
   if (particle[i].class == 0)
                                                                                       35
                                                                                       36
      {
                                                                                       37
        zdisp[j] = i;
        zblock[j] = 1;
                                                                                       38
                                                                                       39
         j++;
      }
                                                                                       40
                                                                                       41
                                                                                       42
/* create datatype for class zero particles */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
                                                                                       43
                                                                                       44
/* prepend particle count */
                                                                                       45
                                                                                       46
MPI_Address(&j, zzdisp);
                                                                                       47
MPI_Address(particle, zzdisp+1);
                                                                                       48
zztype[0] = MPI_INT;
```

```
1
     zztype[1] = Zparticles;
\mathbf{2}
     MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);
3
4
     MPI_Type_commit( &Ztype);
\mathbf{5}
     MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
6
7
8
            /* A probably more efficient way of defining Zparticles */
9
10
     /* consecutive particles with index zero are handled as one block */
11
     i=0;
12
     for (i=0; i < 1000; i++)</pre>
13
        if (particle[i].index == 0)
14
           ſ
15
               for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);</pre>
16
               zdisp[j] = i;
17
               zblock[j] = k-i;
18
               j++;
19
               i = k;
20
            }
21
     MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
22
23
^{24}
                      /* 4.3:
25
                send the first two coordinates of all entries */
26
27
     MPI_Datatype Allpairs;
                                   /* datatype for all pairs of coordinates */
28
29
     MPI_Aint sizeofentry;
30
^{31}
     MPI_Type_extent( Particletype, &sizeofentry);
32
33
          /* sizeofentry can also be computed by subtracting the address
34
              of particle[0] from the address of particle[1] */
35
36
     MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
37
     MPI_Type_commit( &Allpairs);
38
     MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);
39
40
           /* an alternative solution to 4.3 */
41
42
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
43
                                 the extent of one particle entry */
44
     MPI_Aint disp2[3];
45
     MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
46
     int blocklen2[3] = {1, 2, 1};
47
48
     MPI_Address( particle, disp2);
```

```
1
MPI_Address( particle[0].d, disp2+1);
                                                                                       \mathbf{2}
MPI_Address( particle+1, disp2+2);
                                                                                       3
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;</pre>
                                                                                       4
                                                                                       5
MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
                                                                                       6
                                                                                       7
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
                                                                                       8
                                                                                       9
                                                                                       10
                                                                                       11
Example 4.18 The same manipulations as in the previous example, but use absolute
                                                                                       12
addresses in datatypes.
                                                                                       13
                                                                                       14
struct Partstruct
                                                                                       15
   ſ
                                                                                       16
      int class;
                                                                                       17
      double d[6];
                                                                                       18
       char b[7];
                                                                                       19
   };
                                                                                       20
                                                                                       21
struct Partstruct particle[1000];
                                                                                       22
                                                                                       23
            /* build datatype describing first array entry */
                                                                                       ^{24}
                                                                                       25
MPI_Datatype Particletype;
                                                                                       26
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
                                                                                       27
int
              block[3] = \{1, 6, 7\};
                                                                                       28
MPI_Aint
              disp[3];
                                                                                       29
                                                                                       30
MPI_Address( particle, disp);
                                                                                       31
MPI_Address( particle[0].d, disp+1);
                                                                                       32
MPI_Address( particle[0].b, disp+2);
                                                                                       33
MPI_Type_struct( 3, block, disp, type, &Particletype);
                                                                                       34
                                                                                       35
/* Particletype describes first array entry -- using absolute
                                                                                       36
   addresses */
                                                                                       37
                                                                                       38
                    /* 5.1:
                                                                                       39
             send the entire array */
                                                                                       40
                                                                                       41
MPI_Type_commit( &Particletype);
                                                                                       42
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
                                                                                       43
                                                                                       44
                                                                                       45
                   /* 5.2:
                                                                                       46
          send the entries of class zero,
                                                                                       47
          preceded by the number of such entries */
                                                                                       48
```

```
1
\mathbf{2}
     MPI_Datatype Zparticles, Ztype;
3
4
     MPI_Aint
                   zdisp[1000];
\mathbf{5}
                   zblock[1000], i, j, k;
     int
6
     int
                   zzblock[2] = {1,1};
7
     MPI_Datatype zztype[2];
8
     MPI_Aint
                   zzdisp[2];
9
10
     j=0;
11
     for (i=0; i < 1000; i++)
         if (particle[i].index == 0)
12
13
            {
14
               for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);</pre>
15
               zdisp[j] = i;
16
               zblock[j] = k-i;
17
               j++;
18
               i = k;
19
            }
20
     MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
21
     /* Zparticles describe particles with class zero, using
22
        their absolute addresses*/
23
^{24}
     /* prepend particle count */
25
     MPI_Address(&j, zzdisp);
26
     zzdisp[1] = MPI_BOTTOM;
27
     zztype[0] = MPI_INT;
28
     zztype[1] = Zparticles;
29
     MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);
30
^{31}
     MPI_Type_commit( &Ztype);
32
     MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
33
34
35
     Example 4.19 Handling of unions.
36
37
     union {
38
        int
                 ival;
39
        float
                 fval;
40
            } u[1000];
41
42
              utype;
     int
43
44
     /* All entries of u have identical type; variable
45
        utype keeps track of their current type */
46
47
     MPI_Datatype
                     type[2];
48
```

```
1
int
                blocklen[2] = \{1,1\};
                                                                                        \mathbf{2}
MPI_Aint
                disp[2];
                                                                                        3
MPI_Datatype
                mpi_utype[2];
MPI_Aint
                                                                                        4
                i,j;
                                                                                        5
                                                                                        6
/* compute an MPI datatype for each possible union type;
                                                                                        7
   assume values are left-aligned in union storage. */
                                                                                        8
MPI_Address( u, &i);
                                                                                        9
                                                                                        10
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
                                                                                        11
type[1] = MPI_UB;
                                                                                       12
                                                                                        13
                                                                                       14
type[0] = MPI_INT;
                                                                                        15
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);
                                                                                        16
                                                                                        17
type[0] = MPI_FLOAT;
                                                                                       18
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);
                                                                                       19
                                                                                       20
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
                                                                                       21
/* actual communication */
                                                                                       22
                                                                                       23
                                                                                       24
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
                                                                                       25
                                                                                       26
Example 4.20 This example shows how a datatype can be decoded. The routine
                                                                                       27
printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
                                                                                       28
datatypes that are not predefined.
                                                                                       29
                                                                                       30
/*
                                                                                        31
  Example of decoding a datatype.
                                                                                        32
                                                                                       33
  Returns 0 if the datatype is predefined, 1 otherwise
                                                                                       34
 */
#include <stdio.h>
                                                                                       35
                                                                                       36
#include <stdlib.h>
                                                                                       37
#include "mpi.h"
                                                                                       38
int printdatatype( MPI_Datatype datatype )
                                                                                       39
{
                                                                                        40
    int *array_of_ints;
                                                                                       41
    MPI_Aint *array_of_adds;
                                                                                       42
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
                                                                                       43
                                                                                       44
    int i;
                                                                                        45
                                                                                        46
    MPI_Type_get_envelope( datatype,
                                                                                        47
                              &num_ints, &num_adds, &num_dtypes, &combiner );
                                                                                        48
    switch (combiner) {
```

```
1
         case MPI_COMBINER_NAMED:
2
             printf( "Datatype is named:" );
3
             /* To print the specific type, we can match against the
4
                 predefined forms. We can NOT use a switch statement here
5
                 We could also use MPI_TYPE_GET_NAME if we prefered to use
6
                 names that the user may have changed.
7
               */
8
                      (datatype == MPI_INT)
                                                 printf( "MPI_INT\n" );
             if
9
              else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
10
              ... else test for other types ...
11
             return 0;
12
             break;
         case MPI_COMBINER_STRUCT:
13
14
         case MPI_COMBINER_STRUCT_INTEGER:
15
             printf( "Datatype is struct containing" );
16
             array_of_ints
                               = (int *)malloc( num_ints * sizeof(int) );
17
              array_of_adds
18
                          (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
19
              array_of_dtypes = (MPI_Datatype *)
20
                  malloc( num_dtypes * sizeof(MPI_Datatype) );
21
             MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
22
                                array_of_ints, array_of_adds, array_of_dtypes );
23
             printf( " %d datatypes:\n", array_of_ints[0] );
24
             for (i=0; i<array_of_ints[0]; i++) {</pre>
25
                  printf( "blocklength %d, displacement %ld, type:\n",
26
                          array_of_ints[i+1], array_of_adds[i] );
27
                  if (printdatatype( array_of_dtypes[i] )) {
28
                      /* Note that we free the type ONLY if it
29
                         is not predefined */
30
                      MPI_Type_free( &array_of_dtypes[i] );
31
                  }
32
             }
33
             free( array_of_ints );
34
             free( array_of_adds );
35
             free( array_of_dtypes );
36
             break;
37
              ... other combiner values ...
38
         default:
39
             printf( "Unrecognized combiner type\n" );
40
         }
41
         return 1;
42
     }
43
44
           Pack and Unpack
     4.2
45
46
```

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)							
	IN	inbuf	input buffer start (choice)	14 15			
			-	16			
	IN	incount	number of input data items (non-negative integer)	17			
	IN	datatype	datatype of each input data item (handle)	18			
	OUT	outbuf	output buffer start (choice)	19			
	IN	outsize	output buffer size, in bytes (non-negative integer)	20			
	INOUT	position	current position in buffer, in bytes (integer)	21			
		•	- · · · · · · · · · · · · · · · · · · ·	22			
	IN	comm	communicator for packed message (handle)	23			
				24 25 ticket 140.			
-	int MPI_P	<pre>nt MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype,</pre>					
		void *outbuf, int outsize, int *position, MPI_Comm comm)		$^{26}_{27}$ ticket-248T.			
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)							
			28 29				
		Γ(IN) :: inbuf	30				
	TYPE(31					
INTEGER, INTENT(IN) :: incount, outsize TYPE(MPI_Datatype), INTENT(IN) :: datatype							
							INTEGER, INTENT(INOUT) :: position
TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
	INIEG	ER, OFIIONAL, INTENI(001)					
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERRO				37			
<type> INBUF(*), OUTBUF(*)</type>							
	INTEG	ISIZE, POSITION, COMM, IERROR	39				
	40 41						
<pre>{void MPI::Datatype::Pack(const void* inbuf, int incount, void *outbuf,</pre>							
							Dooler
Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer							

Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer ⁴⁴ space specified by outbuf and outsize. The input buffer can be any communication buffer ⁴⁵ allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize ⁴⁶ bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were ⁴⁷ a communication buffer for a message of type MPI_PACKED). ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10 11

The input value of **position** is the first location in the output buffer to be used for packing. **position** is incremented by the size of the packed message, and the output value of **position** is the first location in the output buffer following the locations occupied by the packed message. The **comm** argument is the communicator that will be subsequently used for sending the packed message.

```
7
      MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
8
9
        IN
                    inbuf
                                                    input buffer start (choice)
10
        IN
                    insize
                                                    size of input buffer, in bytes (non-negative integer)
11
        INOUT
                    position
                                                    current position in bytes (integer)
12
13
        OUT
                   outbuf
                                                    output buffer start (choice)
14
        IN
                                                    number of items to be unpacked (integer)
                   outcount
15
        IN
                   datatype
                                                    datatype of each output data item (handle)
16
17
        IN
                                                    communicator for packed message (handle)
                   comm
18
```

int MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf,

```
ticket140. <sup>19</sup>
```

```
int outcount, MPI_Datatype datatype, MPI_Comm comm)
ticket-248T. 21
               MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
          22
                             ierror) BIND(C)
          23
                   TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
          24
                   TYPE(*), DIMENSION(..) :: outbuf
          25
                   INTEGER, INTENT(IN) :: insize, outcount
          26
                    INTEGER, INTENT(INOUT) :: position
          27
                   TYPE(MPI_Datatype), INTENT(IN) :: datatype
          28
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          29
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          30
          ^{31}
```

const(binding deprecated, see Section 15.2)

36 37

38

39 Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from 40 the buffer space specified by inbuf and insize. The output buffer can be any communication 41 buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize 42bytes, starting at address inbuf. The input value of position is the first location in the input 43 buffer occupied by the packed message. position is incremented by the size of the packed 44 message, so that the output value of **position** is the first location in the input buffer after 45 the locations occupied by the message that was unpacked. comm is the communicator used 46 to receive the packed message. 47

int outcount, int& position, const MPI::Comm& comm)

48

1

 $\mathbf{2}$

3

4

5

and comm.

Advice to users. Note the difference between MPI_RECV and MPI_UNPACK: in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of **position** that was output by the previous call, and the same values for **outbuf**, **outcount** and **comm**. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK , where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

Rationale. The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

 31

 $46 \\ 47$

```
1
                 MPI_PACK_SIZE(incount, datatype, comm, size)
            2
                   IN
                            incount
                                                        count argument to packing call (non-negative integer)
            3
                   IN
                            datatype
                                                        datatype argument to packing call (handle)
            4
            5
                   IN
                                                        communicator argument to packing call (handle)
                             comm
            6
                   OUT
                            size
                                                         upper bound on size of packed message, in bytes (non-
            7
                                                         negative integer)
            8
            9
                 int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
            10
                                int *size)
ticket-248T. ^{11}
           12
                 MPI_Pack_size(incount, datatype, comm, size, ierror) BIND(C)
                      INTEGER, INTENT(IN) :: incount
           13
           14
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           15
                     TYPE(MPI_Comm), INTENT(IN) :: comm
            16
                     INTEGER, INTENT(OUT) :: size
            17
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            18
                 MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
           19
                     INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
           20
           21
                 {int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm)
           22
                                const(binding deprecated, see Section 15.2)
           23
                     A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound
           24
                 on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype,
           25
  ticket265. 26
                 outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed
                 by the size parameter, then MPI_PACK_SIZE sets the value of size to MPI_UNDEFINED.
           27
           28
                      Rationale. The call returns an upper bound, rather than an exact bound, since the
           29
                      exact amount of space needed to pack the message may depend on the context (e.g.,
           30
                      first message packed in a packing unit may take more space). (End of rationale.)
           ^{31}
           32
                 Example 4.21 An example using MPI_PACK.
           33
           34
                 int
                             position, i, j, a[2];
           35
                             buff[1000];
                 char
           36
           37
                 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
           38
                 if (myrank == 0)
           39
                 {
           40
                    /* SENDER CODE */
           41
           42
                    position = 0;
           43
                    MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
           44
                    MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
           45
                    MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
           46
                 3
           47
                 else /* RECEIVER CODE */
           48
                    MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

```
1
Example 4.22 An elaborate example.
                                                                                       \mathbf{2}
int
      position, i;
                                                                                       3
float a[1000];
                                                                                       4
char buff[1000];
                                                                                       5
                                                                                       6
MPI_Comm_rank(MPI_Comm_world, &myrank);
                                                                                       7
if (myrank == 0)
                                                                                       8
{
                                                                                       9
  /* SENDER CODE */
                                                                                       10
                                                                                       11
  int len[2];
                                                                                       12
  MPI_Aint disp[2];
                                                                                       13
  MPI_Datatype type[2], newtype;
                                                                                       14
                                                                                       15
  /* build datatype for i followed by a[0]...a[i-1] */
                                                                                       16
                                                                                       17
  len[0] = 1;
                                                                                       18
  len[1] = i;
                                                                                       19
  MPI_Address( &i, disp);
                                                                                       20
  MPI_Address( a, disp+1);
                                                                                       21
  type[0] = MPI_INT;
                                                                                       22
  type[1] = MPI_FLOAT;
                                                                                       23
  MPI_Type_struct( 2, len, disp, type, &newtype);
                                                                                       24
  MPI_Type_commit( &newtype);
                                                                                       25
                                                                                       26
  /* Pack i followed by a[0]...a[i-1]*/
                                                                                       27
                                                                                       28
  position = 0;
                                                                                       29
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
                                                                                       30
                                                                                       31
  /* Send */
                                                                                       32
                                                                                       33
  MPI_Send( buff, position, MPI_PACKED, 1, 0,
                                                                                       34
             MPI_COMM_WORLD);
                                                                                       35
                                                                                       36
/* ****
                                                                                       37
   One can replace the last three lines with
                                                                                       38
   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
                                                                                       39
   **** */
                                                                                       40
}
                                                                                       41
else if (myrank == 1)
                                                                                       42
{
                                                                                       43
   /* RECEIVER CODE */
                                                                                       44
                                                                                       45
  MPI_Status status;
                                                                                       46
                                                                                       47
  /* Receive */
                                                                                       48
```

```
1
\mathbf{2}
       MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
3
4
       /* Unpack i */
5
6
       position = 0;
7
       MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
8
9
       /* Unpack a[0]...a[i-1] */
10
       MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
11
     }
12
13
     Example 4.23 Each process sends a count, followed by count characters to the root; the
14
     root concatenates all characters into one string.
15
16
     int count, gsize, counts[64], totalcount, k1, k2, k,
17
          displs[64], position, concat_pos;
18
     char chr[100], *lbuf, *rbuf, *cbuf;
19
20
     MPI_Comm_size(comm, &gsize);
21
     MPI_Comm_rank(comm, &myrank);
22
           /* allocate local pack buffer */
23
^{24}
     MPI_Pack_size(1, MPI_INT, comm, &k1);
25
     MPI_Pack_size(count, MPI_CHAR, comm, &k2);
26
     k = k1 + k2;
27
     lbuf = (char *)malloc(k);
28
29
           /* pack count, followed by count characters */
30
     position = 0;
^{31}
     MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
32
     MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
33
34
     if (myrank != root) {
35
           /* gather at root sizes of all packed messages */
36
        MPI_Gather( &position, 1, MPI_INT, NULL, 0,
37
                   MPI_DATATYPE_NULL, root, comm);
38
39
           /* gather at root packed messages */
40
        MPI_Gatherv( lbuf, position, MPI_PACKED, NULL,
41
                   NULL, NULL, NULL, root, comm);
42
43
     } else {
                /* root code */
44
           /* gather sizes of all packed messages */
45
        MPI_Gather( &position, 1, MPI_INT, counts, 1,
46
                   MPI_INT, root, comm);
47
48
           /* gather all packed messages */
```

```
displs[0] = 0;
  for (i=1; i < gsize; i++)
     displs[i] = displs[i-1] + counts[i-1];
  totalcount = displs[gsize-1] + counts[gsize-1];
  rbuf = (char *)malloc(totalcount);
  cbuf = (char *)malloc(totalcount);
  MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
            counts, displs, MPI_PACKED, root, comm);
       /* unpack all messages and concatenate strings */
  concat_pos = 0;
  for (i=0; i < gsize; i++) {</pre>
      position = 0;
     MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, &count, 1, MPI_INT, comm);
     MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, cbuf+concat_pos, count, MPI_CHAR, comm);
      concat_pos += count;
  }
   cbuf[concat_pos] = '\0';
}
```

4.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the "external32" data format specified in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but currently the only valid value of the datarep argument is "external32."

Advice to users. These functions could be used, for example, to send typed data in a portable format from one MPI implementation to another. (*End of advice to users.*)

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message and further specifies the exact format of the data. Since MPI_PACK may (and is allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed with MPI_PACK_EXTERNAL. (*End of rationale.*)

1

2

3

4

5 6

7

8

9 10

11

12 13

14 15

16 17

18

19

20

21 22 23

24 25

26

27

28

29 30

31

32

33

34 35

36

37

38

1	¹ MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, posi					
2	IN	datarep	data representation (string)			
3	IN	inbuf	input buffer start (choice)			
5	IN	incount	number of input data items (integer)			
6	IN	datatype	datatype of each input data item (handle)			
7	OUT	outbuf	output buffer start (choice)			
9	IN	outsize	output buffer size, in bytes (integer)			
10	INOUT	position	current position in buffer, in bytes (integer)			
11	moor	position	current position in builer, in bytes (integer)			
ticket140. ¹² ticket140. ₁₃ ticket140. ₁₄	int MPI_P		<pre>*datarep, const void *inbuf, int incount, e, void *outbuf, MPI_Aint outsize,</pre>			
ticket-248T. $^{15}_{16}$	MPI_Pack_	external(datarep, inbuf,	incount, datatype, outbuf, outsize,			
17	position, ierror) BIND(C)					
18 19	CHARACTER(LEN=*), INTENT(IN) :: datarep					
20	TYPE(*), DIMENSION(), INTENT(IN) :: inbuf TYPE(*), DIMENSION() :: outbuf					
21	INTEGER, INTENT(IN) :: incount					
22	TYPE(MPI_Datatype), INTENT(IN) :: datatype					
23	<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
24 25						
26						
27	MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,					
28	TNTFG	POSITION, IERROR) ER INCOUNT, DATATYPE, IE	RROR			
29 30) OUTSIZE, POSITION					
31						
32	<type> INBUF(*), OUTBUF(*)</type>					
33	{void MPI	::Datatype::Pack_external	l(const char* datarep, const void* inbuf,			
34			utbuf, MPI::Aint outsize,			
35 36		MPI::Aint& position)	<pre>const(binding deprecated, see Section 15.2) }</pre>			
37						
38						
39						
40 41						
41 42						
43						
44						
45						
46 47						
48						

MPI_UNF	PACK_EXTERNAL(dat	arep, inbuf, insize, position, outbuf, outsize, position $)$	1
IN	datarep	data representation (string)	2 3
IN	inbuf	input buffer start (choice)	4
IN	insize	input buffer size, in bytes (integer)	5
INOUT	position	current position in buffer, in bytes (integer)	6
OUT	outbuf	output buffer start (choice)	7
		-	8 9
IN	outcount	number of output data items (integer)	10
IN	datatype	datatype of output data item (handle)	11
int MPI_	MPI_Aint insi	nst char *datarep, const void *inbuf, ze, MPI_Aint *position, void *outbuf, MPI_Datatype datatype)	$^{12}_{13}$ ticket140. $^{14}_{14}$ ticket140. 15 ticket-248T.
MPI_Unpa	ck_external(datare	p, inbuf, insize, position, outbuf, outcount,	16
	datatype, ier		17
	ACTER(LEN=*), INTE	NT(IN) :: datarep , INTENT(IN) :: inbuf	18 19
TYPE	20		
INTE	21		
INTE	22		
	GER, INTENT(IN) ::		23
		TENT(IN) :: datatype	24
INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror	25 26
MPI_UNPA	CK_EXTERNAL (DATARE	P, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,	27
	DATATYPE, IER		28
	GER OUTCOUNT, DATA		29
	GER(KIND=MP1_ADDRE ACTER*(*) DATAREP	SS_KIND) INSIZE, POSITION	30
	e> INBUF(*), OUTBU	F(*)	31
• •			32 33
{void MF		k_external(const char* datarep,	34
		<pre>nbuf, MPI::Aint insize, MPI::Aint& position, int outcount) const (binding deprecated, see</pre>	35
	Section 15.2 }	int outcounty const (binaing acpretated, see	36
			37
			38
MPI_PAC	39 40		
IN	datarep	data representation (string)	40
IN	incount	number of input data items (integer)	42
IN		- (, ,	43
	datatype	datatype of each input data item (handle)	44
OUT	size	output buffer size, in bytes (integer)	45
			46
int MPL_		<pre>(const char *datarep, int incount, datatype, MPI_Aint *size)</pre>	47 ticket 140.

Unofficial Draft for Comment Only

ticket-248T.

```
1
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror) BIND(C)
\mathbf{2}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
4
          INTEGER, INTENT(IN) :: incount
          CHARACTER(LEN=*), INTENT(IN) :: datarep
5
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\overline{7}
8
     MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
9
          INTEGER INCOUNT, DATATYPE, IERROR
10
          INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
11
          CHARACTER*(*) DATAREP
12
     {MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep,
13
                    int incount) const(binding deprecated, see Section 15.2) }
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Chapter 5

Collective Communication

5.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- MPI_BARRIER, MPI_IBARRIER: Barrier synchronization across all members of a group (Section 5.3 and Section 5.12.1).
- MPI_BCAST, MPI_IBCAST: Broadcast from one member to all members of a group (Section 5.4 and Section 5.12.2). This is shown as "broadcast" in Figure 5.1.
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV: Gather data from all members of a group to one member (Section 5.5 and Section 5.12.3). This is shown as "gather" in Figure 5.1.
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV: Scatter data from one member to all members of a group (Section 5.6 and Section 5.12.4). This is shown as "scatter" in Figure 5.1.
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7 and Section 5.12.5). This is shown as "allgather" in Figure 5.1.
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 5.8 and Section 5.12.6). This is shown as "complete exchange" in Figure 5.1.
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 5.9.6 and Section 5.12.8) and a variation where the result is returned to only one member (Section 5.9 and Section 5.12.7).
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER: A combined reduction and scatter operation (Section 5.10, Section 5.12.9, and Section 5.12.10).

10 11 1213 14151617 18 ¹⁹ ticket109. ²⁰ ticket109. 21₂₂ ticket109. ₂₃ ticket109. 24 ticket 109. 25 ticket 109. 26 ticket 109. 27²⁸ ticket109. 29 ticket109. 30 ticket 109. 31 ₃₂ ticket109. ₃₃ ticket109. $_{34}$ ticket 109. 35 ticket 109. 36 ticket 109. 37 ticket 109. 38 ticket 109. 39 ⁴⁰ ticket109. 41 ticket109. ⁴² ticket109. 43 ticket 109. 44 ticket109. $_{46}$ ticket 109. 47 ticket109. 48

1 2 3

> 5 6

• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 5.11, Section 5.11.2, Section 5.12.11, and Section 5.12.12).

ticket109. ticket109. ticket109.

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the oper-6 ation. This is discussed further in Section 5.2. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 4. Several collective routines such as broadcast 10 and gather have a single originating or receiving process. Such a process is called the *root*. 11 Some arguments in the collective functions are specified as "significant only at root," and 12are ignored for all participants except the root. The reader is referred to Chapter 4 for 13 information concerning communication buffers, general datatypes and type matching rules, 14and to Chapter 6 for information on how to define groups and create communicators. 15

The type-matching conditions for the collective operations are more strict than the cor-16responding conditions between sender and receiver in point-to-point. Namely, for collective 17operations, the amount of data sent must exactly match the amount of data specified by 18 the receiver. Different type maps (the layout in memory, see Section 4.1) between sender and receiver are still allowed.

Collective [routine calls] operations can (but are not required to) [return] complete as soon as [their] the caller's participation in the collective communication is [complete] finished. A blocking operation is complete as soon as the call returns. A nonblocking (immediate) call requires a separate completion call (cf. Section 3.7). The completion of a [call]collective operation indicates that the caller is [now] free to modify locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise implied by the description of the operation). [Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function Thus, a collective communication operation may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier operation.

Collective communication calls may use the same communicators as point-to-point 32 communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. The collective operations do not have a message tag argument. A more detailed discussion of correct use of collective routines is found in Section 5.13. 36

Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

42The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

The collective operations do not accept a message tag argument. If future revisions of 45MPI define nonblocking collective functions, then tags (or a similar mechanism) might 46need to be added so as to allow the dis-ambiguation of multiple, pending, collective 47 operations.] (End of rationale.) 48

1

2

3

4

5

7

8

9

19 20ticket109. 21 ticket109. 22 ticket109. 23 ticket109. 24 ticket109. $\frac{1}{25}$ ticket109. 26ticket109. ticket 109. 27 2829 30 31 33 34 ticket109. 35 37 38 39 40 41 ticket109. 43 44

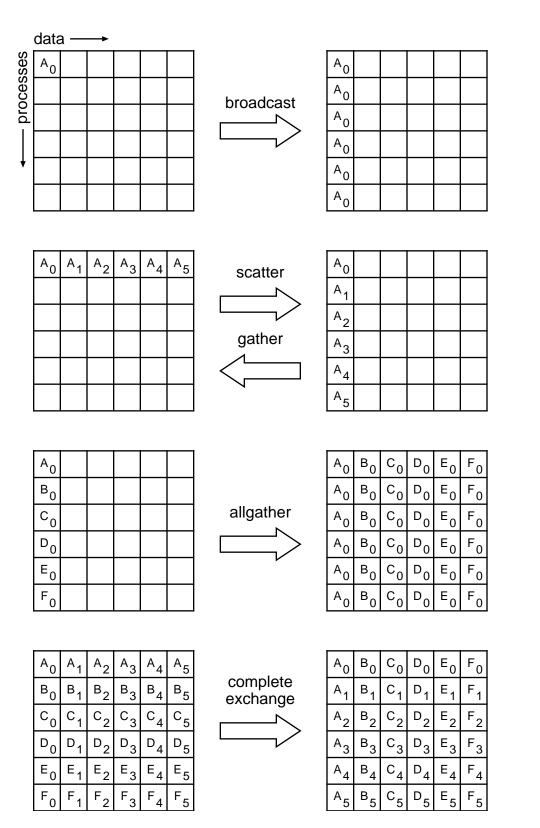


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 31

 41

 $44 \\ 45$

Advice to users. It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.

On the other hand, a correct, portable program must allow for the fact that a collective call may be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 5.13. (End of advice to users.)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.13. (End of advice to implementors.)

17 18 19

20

21

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are *not* correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

22 23 24

25

29

33 34

38

39

40

41

5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating 26processes. The routines do not have group identifiers as explicit arguments. Instead, there 27is a communicator argument. Groups and communicators are discussed in full detail in 28Chapter 6. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: intra-communicators and inter-communicators. An intracommunicator 30 can be thought of as an indentifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context. 32

ticket109. 31

5.2.1 Specifics for Intracommunicator Collective Operations

35 All processes in the group identified by the intracommunicator must call the collective 36 routine. 37

In many cases, collective communication can occur "in place" for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

42Rationale. The "in place" operations are provided to reduce unnecessary memory 43 motion by both the MPI implementation and by the user. Note that while the simple 44check of testing whether the send and receive buffers have the same address will 45work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., 46MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits 47 aliasing of arguments; the approach of using a special value to denote "in place" 48 operation eliminates that difficulty. (End of rationale.)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

Advice to users. By allowing the "in place" option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has. Some intracommunicator collective operations do not support the "in place" option (e.g., MPI_ALLTOALLV).] (End of advice to users.)

5.2.2 Applying Collective Operations to Intercommunicators

To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [56]:

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER
- MPI_BARRIER, MPI_IBARRIER

All-To-One All processes contribute to the result. One process receives the result.

- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV
- MPI_REDUCE, MPI_IREDUCE

One-To-All One process contributes to the result. All processes receive the result.

- MPI_BCAST, MPI_IBCAST
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV

Other Collective operations that do not fit into one of the above categories.

• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN

The data movement patterns of MPI_SCAN, MPI_ISCAN [and], MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy.

The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.

The following collective operations also apply to intercommunicators:

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$ 8

9

10

11

1213

14

¹⁵ ticket109.

¹⁶ ticket109. 17 ticket 109.

 18 ticket 109.

¹⁹ ticket109.

20 ticket109.

21 ticket109.

 $_{22}$ ticket 109.

₂₃ ticket109.

₂₆ ticket109. ₂₇ ticket109.

 $_{28}$ ticket 109.

 30 ticket 109.

 $_{35}$ ticket 109.

³⁷ ticket109.

 38 ticket 109.

ticket109. ³⁶ ticket109.

ticket109. 32 ticket 109.

 24

25

29

31

33

39

40 41

42

43

44

4546

47

48

ticket109.

17

18

19 20 21

22

23

 24

25 26

27 28 29

30

 31

32

33

34 35

36 37

- MPI_BARRIER, MPI_IBARRIER
- MPI_BCAST, MPI_IBCAST
 - MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,
 - MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV,
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV,
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW,
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE,
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER.

In C++, the bindings for these functions are in the MPI::Comm class. However, since the collective operations do not make sense on a C++ MPI::Comm (as it is neither an intercommunicator nor an intracommunicator), the functions are all pure virtual.

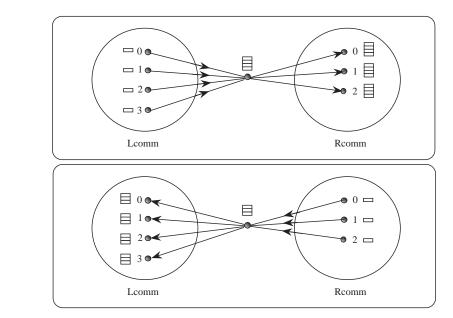


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

39 40 41

42

38

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective
 routine.

Note that the "in place" option for intracommunicators does not apply to intercommunicators since in the intercommunicator case there is no communication from a process
 to itself.

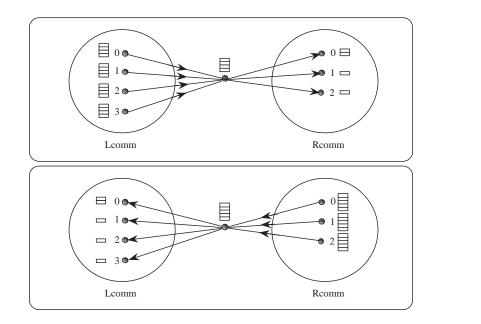


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

For intercommunicator collective communication, if the operation is in the All-To-One or One-To-All categories, then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. In this case, for the group containing the root process, all processes in the group must call the routine using a special argument for the root. For this, the root process uses the special root value MPI_ROOT; all other processes in the same group as the root use MPI_PROC_NULL. All processes in the other group (the group that is the remote group relative to the root process) must call the collective routine and provide the rank of the root. If the operation is in the All-To-All category, then the transfer is bidirectional.

Rationale. Operations in the All-To-One and One-To-All categories are unidirectional by nature, and there is a clear way of specifying direction. Operations in the All-To-All category will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

Barrier Synchronization

5.3

Z

 24

ticket-248T.

	1	INTEGE	R, OPTION	NAL, INTENT(OUT)	:: ierror				
	2 3	MPI_BARRIE	ER(COMM, 1	IERROR)					
	4	INTEGER COMM, IERROR							
	5	<pre>{void MPI::Comm::Barrier() const = 0(binding deprecated, see Section 15.2) }</pre>							
	6 7			,	PI_BARRIER blocks the caller until all group mem-				
	8	bers have called it. The call returns at any process only after all group members have entered the call.							
	9 10		n is an inte	ercommunicator, MI	PI_BARRIER involves two groups. The call returns				
	11	-	0	x (0 x /	e intercommunicator only after all members of the				
	12	· ·	(0 1 /	/	call (and vice versa). A process may return from oup have entered the call.				
	13 14		ore an proc						
	15	5.4 Broa	adcast						
	16 17								
	18		.		,				
	19			ount, datatype, root	,				
	20 21	INOUT	buffer		starting address of buffer (choice)				
	22	IN	count		number of entries in buffer (non-negative integer)				
	23 24	IN	datatype		data type of buffer (handle)				
	24 25	IN	root		rank of broadcast root (integer)				
	26	IN	comm		communicator (handle)				
	27 28	<pre>int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,</pre>							
ticket-248T.		MPI_Comm comm)							
	30	<pre>MPI_Bcast(buffer, count, datatype, root, comm, ierror) BIND(C)</pre>							
	31 32	TYPE(*), DIMENSION() :: buffer INTEGER, INTENT(IN) :: count, root							
	33	INTEGER, INTENT(IN) :: count, root TYPE(MPI_Datatype), INTENT(IN) :: datatype							
	34	TYPE(MPI_Comm), INTENT(IN) :: comm							
	35 36	INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
	37	MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)							
	38 39	<type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR</type>							
	40			cast(void* buffe					
	41	(voiù mi.			datatype, int root) const = 0(binding				
	42 43		deprec	cated, see Section 1	5.2) }				
	43				PI_BCAST broadcasts a message from the process				
	45		-		oup, itself included. It is called by all members of comm and root. On return, the content of root's				
	46 47		-	other processes.	comm and root. On return, the content of root's				
	48	1		-					

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful here.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1

Broadcast 100 ints from process 0 to every process in the group.

MPI_Comm comm; int array[100]; int root=0; ... MPI_Bcast(array, 100, MPI_INT, root, comm);

As in many of our example code fragments, we assume that some of the variables (such as comm in the above) have been assigned appropriate values.

 24

1 2

47

48

CHAPTER 5. COLLECTIVE COMMUNICATION

5.5 Gather

```
3
            4
                 MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
            5
                   IN
                              sendbuf
            6
                                                          starting address of send buffer (choice)
            7
                   IN
                              sendcount
                                                          number of elements in send buffer (non-negative inte-
            8
                                                          ger)
            9
                   IN
                              sendtype
                                                          data type of send buffer elements (handle)
            10
                   OUT
                              recvbuf
                                                          address of receive buffer (choice, significant only at
           11
                                                          root)
           12
            13
                   IN
                              recvcount
                                                          number of elements for any single receive (non-negative
           14
                                                          integer, significant only at root)
           15
                   IN
                                                          data type of recv buffer elements (significant only at
                              recvtype
            16
                                                          root) (handle)
            17
                   IN
                                                          rank of receiving process (integer)
            18
                              root
            19
                   IN
                              comm
                                                          communicator (handle)
           20
  ticket
140. \frac{1}{22}
           21
                 int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                                 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
           23
                                 MPI_Comm comm)
ticket229.1.<sup>24</sup>
ticket-248T. 25
                 MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
           26
                                 root, comm, ierror) BIND(C)
                      TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
           27
                      TYPE(*), DIMENSION(..) :: recvbuf
           28
                      INTEGER, INTENT(IN) :: sendcount, recvcount, root
           29
                      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           30
                      TYPE(MPI_Comm), INTENT(IN) :: comm
           31
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           32
           33
                 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
           34
                                 ROOT, COMM, IERROR)
           35
                      <type> SENDBUF(*), RECVBUF(*)
           36
                      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
           37
           38
                 {void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
           39
                                 MPI::Datatype& sendtype, void* recvbuf, int recvcount,
            40
                                 const MPI::Datatype& recvtype, int root) const = 0(binding
           41
                                 deprecated, see Section 15.2) }
           42
                     If comm is an intracommunicator, each process (root process included) sends the con-
           43
                 tents of its send buffer to the root process. The root process receives the messages and stores
           44
                 them in rank order. The outcome is as if each of the n processes in the group (including
           45
                 the root process) had executed a call to
           46
```

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

and the root had executed n calls to

```
MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...),
```

where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent().

An alternative description is that the n messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV (recvbuf, recvcount·n, recvtype, ...).

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on each process must be equal to the type signature of recvcount, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the **recvcount** argument at the root indicates the number of items it receives from *each* process, not the total number of items it receives.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root. $\mathbf{2}$

 24

1 MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, $\mathbf{2}$ comm) 3 IN sendbuf starting address of send buffer (choice) 4 IN sendcount number of elements in send buffer (non-negative inte-5ger) 6 7 IN sendtype data type of send buffer elements (handle) 8 OUT recvbuf address of receive buffer (choice, significant only at 9 root) 10 IN non-negative integer array (of length group size) conrecvcounts 11 taining the number of elements that are received from 12each process (significant only at root) 13 IN displs integer array (of length group size). Entry i specifies 14the displacement relative to recvbuf at which to place 15the incoming data from process i (significant only at 1617root) 18 IN recvtype data type of recv buffer elements (significant only at 19root) (handle) 20IN rank of receiving process (integer) root 21IN comm communicator (handle) 22 23int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, ticket140. 24 ticket140. 25 void* recvbuf, const int recvcounts[], const int displs[], ticket140. 26 MPI_Datatype recvtype, int root, MPI_Comm comm) ticket229.1. 27 MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, ticket-248T. 28 recvtype, root, comm, ierror) BIND(C) 29 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 30 TYPE(*), DIMENSION(..) :: recvbuf 31 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root 32 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 33 TYPE(MPI_Comm), INTENT(IN) :: comm 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 36 37 RECVTYPE, ROOT, COMM, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 39 40 COMM, IERROR 41 {void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const 42MPI::Datatype& sendtype, void* recvbuf, 43 const int recvcounts[], const int displs[], 44 const MPI::Datatype& recvtype, int root) const = 0(binding 45deprecated, see Section 15.2 } 46 47MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count 48of data from each process, since recvcounts is now an array. It also allows more flexibility If **comm** is an intracommunicator, the outcome is *as if* each process, including the root process, sends a message to the root,

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...),
```

and the root executes n receives,

```
\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs}[j] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[j], \texttt{recvtype}, i, ...).
```

The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype).

The receive buffer is ignored for all non-root processes.

The type signature implied by sendcount, sendtype on process i must be equal to the type signature implied by recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 5.6.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer[].

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

The examples in this section use intracommunicators.

Example 5.2

Gather 100 ints from every process in group to root. See [f]Figure 5.4.

```
MPI_Comm comm; 41
int gsize,sendarray[100]; 42
int root, *rbuf; 43
...
MPI_Comm_size(comm, &gsize); 45
rbuf = (int *)malloc(gsize*100*sizeof(int)); 46
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm); 47
48
```

1

 $\mathbf{2}$

3

4

5 6

7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

27

28

29

30

31

32

33 34

35 36

37 38

40

³⁹ ticket0.

 $_{26}$ ticket0.

1	100 100 100
2	all processes
3	
4	
5	
6	at root
7	n.
8	rbuf
9	Figure 5.4. The rest presses gethers 100 into from each presses in the group
10	Figure 5.4: The root process gathers 100 ints from each process in the group.
11	
12	Example 5.3
13	Previous example modified – only the root allocates memory for the receive buffer.
14	
15	MPI_Comm comm;
16	<pre>int gsize,sendarray[100];</pre>
17	<pre>int root, myrank, *rbuf;</pre>
18	
19	<pre>MPI_Comm_rank(comm, &myrank);</pre>
20	if (myrank == root) {
21	<pre>MPI_Comm_size(comm, &gsize);</pre>
22	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
23	}
24	<pre>MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);</pre>
25	
26	
27	Example 5.4
28	Do the same as the previous example, but use a derived datatype. Note that the type
29	cannot be the entire set of gsize*100 ints since type matching is defined pairwise between
30	the root and each process in the gather.
31	NDT (
32	MPI_Comm comm;
33	<pre>int gsize,sendarray[100];</pre>
34	<pre>int root, *rbuf;</pre>
35	MPI_Datatype rtype;
36	
37	<pre>MPI_Comm_size(comm, &gsize);</pre>
38	<pre>MPI_Type_contiguous(100, MPI_INT, &rtype);</pre>
39	<pre>MPI_Type_commit(&rtype);</pre>
40	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
41	<pre>MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);</pre>
42	
	Example 5.5
43 44	Example 5.5 New have each process send 100 ints to root, but place each set (of 100) stride ints
	Now have each process send 100 ints to root, but place each set (of 100) stride ints
45 46	apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume $stride \geq 100$. See Figure 5.5.
40 47	Assume surve ≥ 100 . See l'igure 0.0.
47	
-10	

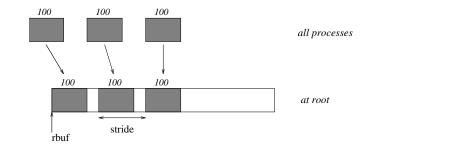


Figure 5.5: The root process gathers 100 ints from each process in the group, each set is placed stride ints apart.

```
MPI_Comm comm;
                                                                                          13
    int gsize,sendarray[100];
                                                                                          14
    int root, *rbuf, stride;
                                                                                          15
    int *displs,i,*rcounts;
                                                                                          16
                                                                                          17
    . . .
                                                                                          18
                                                                                          19
    MPI_Comm_size(comm, &gsize);
                                                                                          20
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                          21
    displs = (int *)malloc(gsize*sizeof(int));
                                                                                          22
    rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                          23
    for (i=0; i<gsize; ++i) {</pre>
                                                                                          ^{24}
        displs[i] = i*stride;
                                                                                          25
        rcounts[i] = 100;
                                                                                          26
    }
                                                                                          27
    MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                                                          28
                                                                        root, comm);
                                                                                          29
                                                                                          30
    Note that the program is erroneous if stride < 100.
                                                                                          ^{31}
                                                                                          32
Example 5.6
                                                                                          33
    Same as Example 5.5 on the receiving side, but send the 100 ints from the 0th column
                                                                                          34
of a 100 \times 150 int array, in C. See Figure 5.6.
                                                                                          35
                                                                                          36
    MPI_Comm comm;
                                                                                          37
    int gsize, sendarray[100][150];
                                                                                          38
    int root, *rbuf, stride;
                                                                                          39
    MPI_Datatype stype;
                                                                                          40
    int *displs,i,*rcounts;
                                                                                          41
                                                                                          42
    . . .
                                                                                          43
                                                                                          44
    MPI_Comm_size(comm, &gsize);
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                          45
                                                                                          46
    displs = (int *)malloc(gsize*sizeof(int));
                                                                                          47
    rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                          48
    for (i=0; i<gsize; ++i) {</pre>
```

Unofficial Draft for Comment Only

1

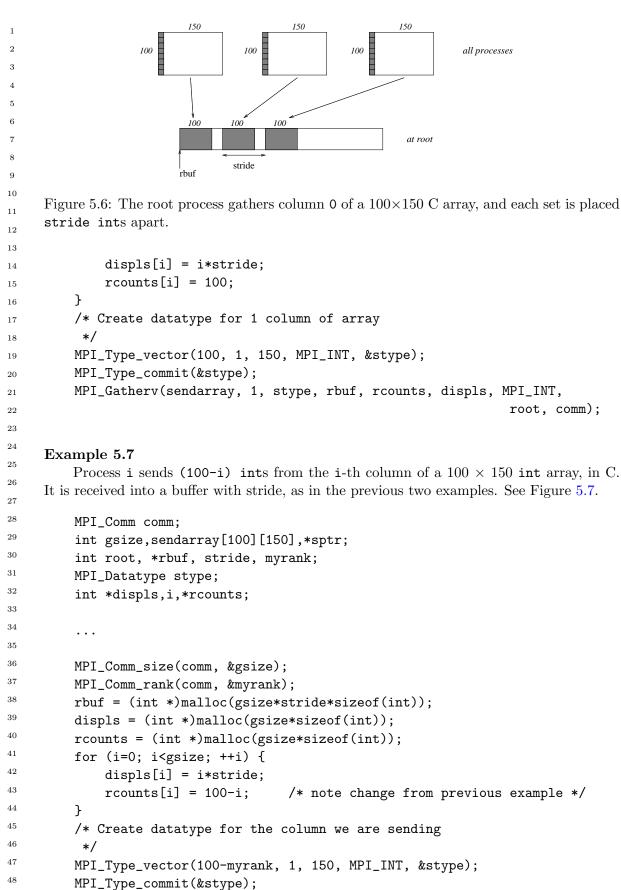
2 3 4

5

6 7

8 9

10



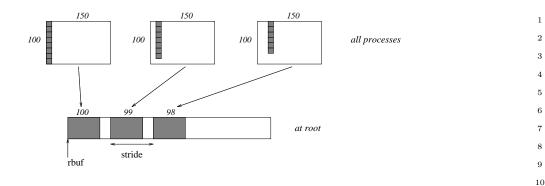


Figure 5.7: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride ints apart.

```
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                     root, comm);
```

Note that a different amount of data is received from each process.

Example 5.8

Same as Example 5.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 4.16, Section 4.1.14.

```
26
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
                                                                                  27
int root, *rbuf, stride, myrank, blocklen[2];
                                                                                  28
                                                                                  29
MPI_Aint disp[2];
                                                                                  30
MPI_Datatype stype,type[2];
                                                                                  31
int *displs,i,*rcounts;
                                                                                  32
                                                                                  33
. . .
                                                                                  34
MPI_Comm_size(comm, &gsize);
                                                                                  35
                                                                                  36
MPI_Comm_rank(comm, &myrank);
                                                                                  37
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
                                                                                  38
                                                                                  39
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                  40
for (i=0; i<gsize; ++i) {</pre>
                                                                                  41
    displs[i] = i*stride;
                                                                                  42
    rcounts[i] = 100-i;
}
                                                                                  43
                                                                                  44
/* Create datatype for one int, with extent of entire row
 */
                                                                                  45
disp[0] = 0;
                                                                                  46
                    disp[1] = 150*sizeof(int);
                                                                                  47
type[0] = MPI_INT; type[1] = MPI_UB;
                                                                                  48
blocklen[0] = 1;
                    blocklen[1] = 1;
```

Unofficial Draft for Comment Only

1 2

3 4 5

6

7

9

11

1213

14

15

16

17

18 19

2021

22

23

24

```
1
         MPI_Type_create_struct(2, blocklen, disp, type, &stype);
\mathbf{2}
         MPI_Type_commit(&stype);
3
         sptr = &sendarray[0][myrank];
4
         MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
5
                                                                           root, comm);
6
7
     Example 5.9
8
         Same as Example 5.7 at sending side, but at receiving side we make the stride between
9
     received blocks vary from block to block. See Figure 5.8.
10
11
         MPI_Comm comm;
12
         int gsize, sendarray[100][150], *sptr;
13
          int root, *rbuf, *stride, myrank, bufsize;
14
         MPI_Datatype stype;
15
         int *displs,i,*rcounts,offset;
16
17
          . . .
18
19
         MPI_Comm_size(comm, &gsize);
20
         MPI_Comm_rank(comm, &myrank);
21
22
         stride = (int *)malloc(gsize*sizeof(int));
23
          . . .
24
         /* stride[i] for i = 0 to gsize-1 is set somehow
25
           */
26
27
         /* set up displs and rcounts vectors first
28
           */
29
         displs = (int *)malloc(gsize*sizeof(int));
30
         rcounts = (int *)malloc(gsize*sizeof(int));
31
         offset = 0;
32
         for (i=0; i<gsize; ++i) {</pre>
33
              displs[i] = offset;
34
              offset += stride[i];
35
              rcounts[i] = 100-i;
36
         }
37
         /* the required buffer size for rbuf is now easily obtained
38
           */
39
         bufsize = displs[gsize-1]+rcounts[gsize-1];
40
         rbuf = (int *)malloc(bufsize*sizeof(int));
41
         /* Create datatype for the column we are sending
42
           */
43
         MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
44
         MPI_Type_commit(&stype);
45
         sptr = &sendarray[0][myrank];
46
         MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
47
                                                                   root, comm);
48
```

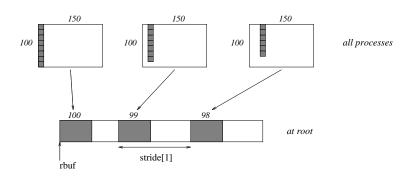


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

Example 5.10

Process i sends num ints from the i-th column of a 100×150 int array, in C. The complicating factor is that the various values of num are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```
19
MPI_Comm comm;
                                                                                 20
int gsize, sendarray[100][150], *sptr;
                                                                                 21
int root, *rbuf, myrank, blocklen[2];
                                                                                 22
MPI_Aint disp[2];
                                                                                 23
MPI_Datatype stype,type[2];
                                                                                  ^{24}
int *displs,i,*rcounts,num;
                                                                                  25
                                                                                  26
. . .
                                                                                  27
                                                                                  28
MPI_Comm_size(comm, &gsize);
                                                                                 29
MPI_Comm_rank(comm, &myrank);
                                                                                  30
                                                                                  31
/* First, gather nums to root
                                                                                  32
 */
                                                                                  33
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                 34
MPI_Gather(&num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
                                                                                 35
/* root now has correct rcounts, using these we set displs[] so
                                                                                 36
 * that data is placed contiguously (or concatenated) at receive end
                                                                                 37
 */
                                                                                  38
displs = (int *)malloc(gsize*sizeof(int));
                                                                                  39
displs[0] = 0;
                                                                                  40
for (i=1; i<gsize; ++i) {</pre>
                                                                                  41
    displs[i] = displs[i-1]+rcounts[i-1];
                                                                                  42
}
                                                                                  43
/* And, create receive buffer
                                                                                  44
 */
                                                                                  45
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
                                                                                  46
                                                              *sizeof(int));
                                                                                  47
/* Create datatype for one int, with extent of entire row
                                                                                  48
```

1

 $\mathbf{2}$

3 4 5

6

9 10

11

12 13

14

15

16

17

```
1
                       */
            \mathbf{2}
                     disp[0] = 0;
                                           disp[1] = 150*sizeof(int);
            3
                     type[0] = MPI_INT; type[1] = MPI_UB;
            4
                                           blocklen[1] = 1;
                     blocklen[0] = 1;
            5
                     MPI_Type_create_struct( 2, blocklen, disp, type, &stype );
            6
                     MPI_Type_commit(&stype);
            7
                     sptr = &sendarray[0][myrank];
            8
                     MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
            9
                                                                                         root, comm);
           10
           11
                 5.6
                      Scatter
           12
           13
           14
           15
                 MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
           16
                   IN
                             sendbuf
                                                         address of send buffer (choice, significant only at root)
           17
                   IN
                             sendcount
                                                         number of elements sent to each process (non-negative
           18
                                                         integer, significant only at root)
           19
                   IN
                             sendtype
                                                         data type of send buffer elements (significant only at
           20
                                                         root) (handle)
           21
           22
                   OUT
                             recvbuf
                                                         address of receive buffer (choice)
           23
                   IN
                                                         number of elements in receive buffer (non-negative in-
                             recvcount
           24
                                                         teger)
           25
                   IN
                             recvtype
                                                         data type of receive buffer elements (handle)
           26
           27
                   IN
                             root
                                                         rank of sending process (integer)
           28
                   IN
                             comm
                                                         communicator (handle)
           29
           30
  ticket140. 31
                 int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
           32
                                MPI_Comm comm)
ticket229.1.<sup>33</sup>
ticket-248T. <sup>34</sup>
                 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
           35
                                root, comm, ierror) BIND(C)
           36
                     TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
           37
                     TYPE(*), DIMENSION(..) :: recvbuf
           38
                     INTEGER, INTENT(IN) :: sendcount, recvcount, root
           39
                     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           40
                     TYPE(MPI_Comm), INTENT(IN) :: comm
           41
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           42
                 MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
           43
                                ROOT, COMM, IERROR)
           44
                      <type> SENDBUF(*), RECVBUF(*)
           45
                      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
           46
           47
                 {void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const
           48
                                MPI::Datatype& sendtype, void* recvbuf, int recvcount,
```

<pre>const MPI::Datatype& recvtype,</pre>	int root)	const = 0	0 <i>(binding</i>
deprecated, see Section 15.2 }			

MPI_SCATTER is the inverse operation to MPI_GATHER.

If comm is an intracommunicator, the outcome is *as if* the root executed n send operations,

 $MPI_Send(sendbuf + i \cdot sendcount \cdot extent(sendtype), sendcount, sendtype, i, ...),$

and each process executed a receive,

MPI_Recv(recvbuf, recvcount, recvtype, i, ...).

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount \cdot n, sendtype, ...). This message is split into n equal segments, the *i*-th segment is sent to the *i*-th process in the group, and each process receives this message as above.

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

- $45 \\ 46$

 24

 31

	2	PI_SCATT	ERV(sendbuf, sendcounts, d comm)	s, displs, sendtype, recvbuf, recvcount, recvtype, root,				
	3 4	IN sendbuf address of send buffer (choice, significant		address of send buffer (choice, significant only at root)				
		Ν	sendcounts	non-negative integer array (of length group size) speci- fying the number of elements to send to each [processor]rank ticket0.				
ticket109.	8 9	Ν	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i				
	10 11	Ν	sendtype	data type of send buffer elements (handle)				
		TUC	recvbuf	address of receive buffer (choice)				
	13 14	Ν	recvcount	number of elements in receive buffer (non-negative in- teger)				
	15 16	N	recvtype	data type of receive buffer elements (handle)				
		N	root	rank of sending process (integer)				
	¹⁸ I	N	comm	communicator (handle)				
<pre>ticket140.²⁰ ticket140.²¹ ticket140.²¹ ticket229.1.²³ ticket229.1.²³ mPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, mPI_Comm co mPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) mPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) mPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, root mPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvtype mPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvtype mPI_Scatterv(sendbuf, sendcounts, sendtype, recvtype mPI_Scatterv(sendbuf, sendcounts, sendtype, recvtype mPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVEOUNT, RECVTYPE, ROOT, COMM, IERROR void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], const MPI::Datatype& recvtype mit root) const = 0(binding deprecated, see Section 15.2) } MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a vary MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a vary MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a vary definition mPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a vary definition mPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a vary definition mPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a vary definition method const method const method proceeding approximation of the proceeding approximat</pre>								
	45 COI 46 mC	count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing an additional argument, displs.						

If comm is an intracommunicator, the outcome is as if the root executed **n** send operations,

```
MPI_Send(sendbuf + displs[i] · extent(sendtype), sendcounts[i], sendtype, i, ...),
```

and each process executed a receive,

MPI_Recv(recvbuf, recvcount, recvtype, i, ...).

The send buffer is ignored for all non-root processes.

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be read more than once.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

The examples in this section use intracommunicators.

Example 5.11

The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

1

 $\mathbf{2}$

3

4 5

6

7 8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31 32

33 34

35 36

37

38

39 40

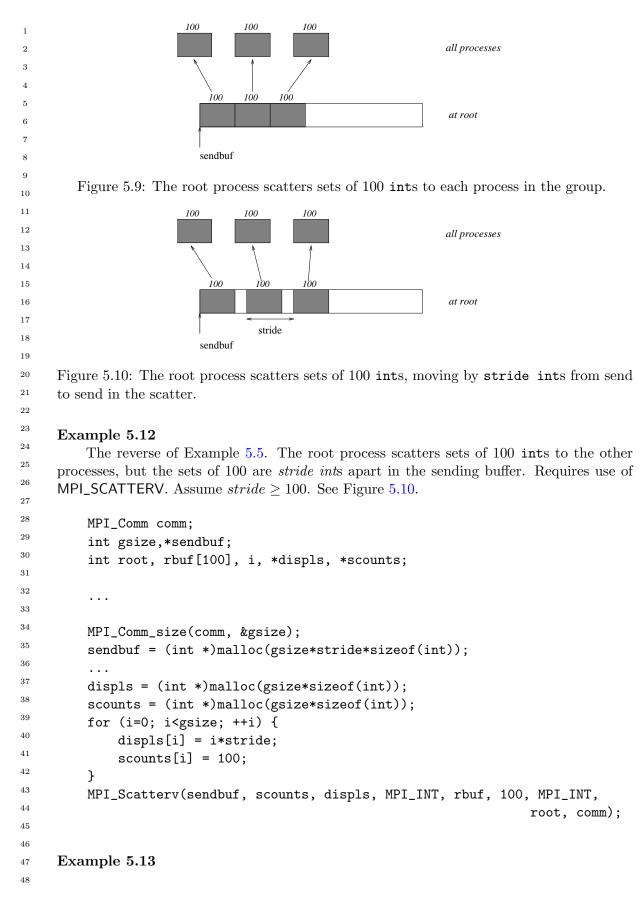
41

42

43

44

45 46 47



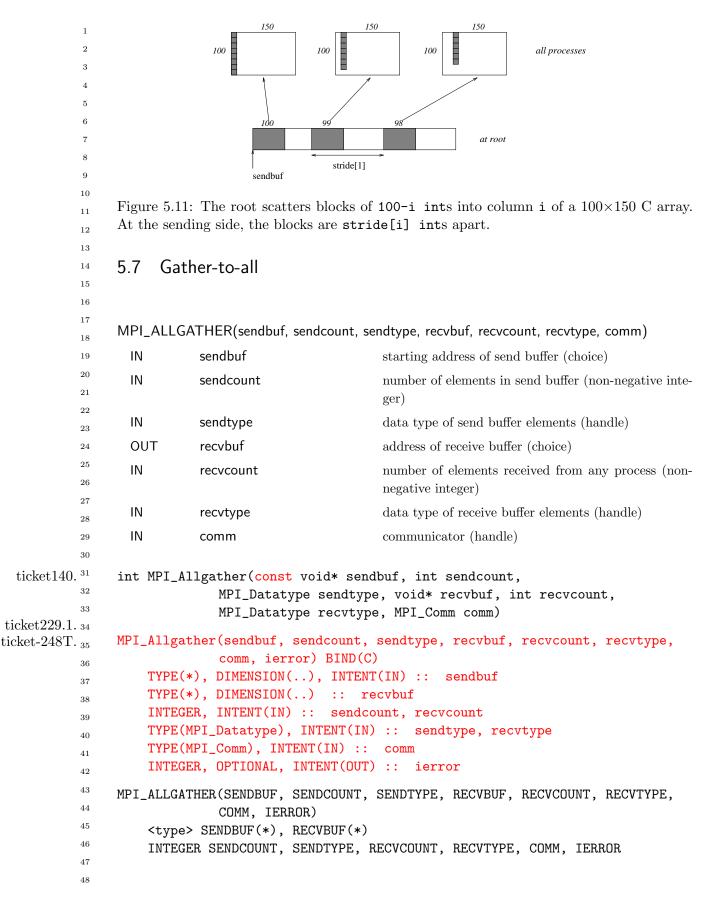
The reverse of Example 5.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the *i*-th column of a 100×150 C array. See Figure 5.11.

```
MPI_Comm comm;
                                                                                    5
int gsize, recvarray[100][150], *rptr;
                                                                                    6
int root, *sendbuf, myrank, *stride;
                                                                                    7
MPI_Datatype rtype;
                                                                                    8
int i, *displs, *scounts, offset;
                                                                                    9
                                                                                    10
. . .
MPI_Comm_size(comm, &gsize);
                                                                                    11
MPI_Comm_rank(comm, &myrank);
                                                                                    12
                                                                                    13
stride = (int *)malloc(gsize*sizeof(int));
                                                                                    14
                                                                                    15
. . .
/* stride[i] for i = 0 to gsize-1 is set somehow
                                                                                    16
 * sendbuf comes from elsewhere
                                                                                    17
 */
                                                                                    18
                                                                                    19
. . .
displs = (int *)malloc(gsize*sizeof(int));
                                                                                    20
scounts = (int *)malloc(gsize*sizeof(int));
                                                                                    21
offset = 0;
                                                                                    22
for (i=0; i<gsize; ++i) {</pre>
                                                                                    23
    displs[i] = offset;
                                                                                    ^{24}
    offset += stride[i];
                                                                                    25
    scounts[i] = 100 - i;
                                                                                    26
}
                                                                                    27
/* Create datatype for the column we are receiving
                                                                                    28
 */
                                                                                    29
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
                                                                                    30
MPI_Type_commit(&rtype);
                                                                                    ^{31}
rptr = &recvarray[0][myrank];
                                                                                    32
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
                                                                                    33
                                                              root, comm);
                                                                                    34
                                                                                    35
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
                                                                                    48
```

1

2

3



<pre>MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER() is as if all processes executed n calls to MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount, recvtype,root,comm) for root = 0 ,, n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER. The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. If comm is an intercommunicator, then each process of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (group B). Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa. Advice to users. The communication pattern of MPI_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount = 0 for the communication</pre>
<pre>MP1_Gather(sendbur,sendcount,sendtype,recvcount, recvtype,root,comm) for root = 0 ,, n-1. The rules for correct usage of MP1_ALLGATHER are easily found from the corresponding rules for MP1_GATHER. The "in place" option for intracommunicators is specified by passing the value MP1_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. If comm is an intercommunicator, then each process of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (group B). Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa. Advice to users. The communication pattern of MP1_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount = 0 for the communication</pre>
for root = 0 ,, n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER. The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. If comm is an intercommunicator, then each process of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (group B). Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa. Advice to users. The communication pattern of MPI_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount = 0 for the communication
in the reverse direction. (End of advice to users.)

	178		CHAPTER 5. COLLECTIVE COMMUNICATION		
1 2	MPI_ALL	GATHERV(sendbuf, s	endcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)		
3	IN	sendbuf	starting address of send buffer (choice)		
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)		
7	IN	sendtype	data type of send buffer elements (handle)		
8	OUT	recvbuf	address of receive buffer (choice)		
9 10	IN	recvcounts	non-negative integer array (of length group size) con-		
11 12			taining the number of elements that are received from each process		
13 14 15	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i		
16 17	IN	recvtype	data type of receive buffer elements (handle)		
18	IN	comm	communicator (handle)		
19 ticket140. 20 ticket140. 21 ticket140. 22 ticket229.1. 23 ticket-248T. 24 25 26 27 28 29 30 31 32 33 34 35 36 37	MPI_Allg TYPE TYPE INTE TYPE INTE MPI_ALLG <typ INTE IERR</typ 	MPI_Datatype const int d: gatherv(sendbuf, s recvtype, ca E(*), DIMENSION(E(*), DIMENSION(EGER, INTENT(IN) : E(MPI_Datatype), I E(MPI_Comm), INTEN EGER, OPTIONAL, IN GATHERV(SENDBUF, S RECVTYPE, Ca De> SENDBUF(*), RE EGER SENDCOUNT, SE ROR	<pre>: sendcount, recvcounts(*), displs(*) NTENT(IN) :: sendtype, recvtype T(IN) :: comm TENT(OUT) :: ierror ENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, DMM, IERROR) CVBUF(*) NDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,</pre>		
38 39 40 41 42	<pre>{void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const</pre>				
43 44 45 46 47	MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes re- ceive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. These blocks need not all be the same size.				

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.

```
If comm is an intracommunicator, the outcome is as if all processes executed calls to
                                                                                             1
                                                                                             \mathbf{2}
    MPI_GATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
                                                                                             3
                                                            recvtype,root,comm),
                                                                                             4
                                                                                             5
for root = 0, ..., n-1. The rules for correct usage of MPI_ALLGATHERV are easily
                                                                                             6
found from the corresponding rules for MPI_GATHERV.
                                                                                             7
    The "in place" option for intracommunicators is specified by passing the value
                                                                                             8
MPI_IN_PLACE to the argument sendbuf at all processes. In such a case, sendcount and
                                                                                             9
sendtype are ignored, and the input data of each process is assumed to be in the area where
                                                                                             10
that process would receive its own contribution to the receive buffer.
                                                                                             11
    If comm is an intercommunicator, then each process of one group (group A) contributes
                                                                                             12
sendcount data items; these data are concatenated and the result is stored at each process
                                                                                             13
in the other group (group B). Conversely the concatenation of the contributions of the
                                                                                             14
processes in group B is stored at each process in group A. The send buffer arguments in
                                                                                             15
group A must be consistent with the receive buffer arguments in group B, and vice versa.
                                                                                             16
                                                                                             17
       Example using MPI_ALLGATHER
5.7.1
                                                                                             18
The example in this section uses intracommunicators.
                                                                                             19
                                                                                             20
Example 5.14
                                                                                            21
    The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100
                                                                                            22
ints from every process in the group to every process.
                                                                                            23
```

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

After the call, every process has the group-wide concatenation of the sets of data.

 24

25

26

27

28

29

30

31 32

 $\mathbf{5}$

CHAPTER 5. COLLECTIVE COMMUNICATION

All-to-All Scatter/Gather 5.8

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

Э		,						
6	IN	sendbuf	starting address of send buffer (choice)					
7 8	IN	sendcount	number of elements sent to each process (non-negative integer)					
9	IN	sendtype	data type of send buffer elements (handle)					
10 11	OUT	recvbuf	address of receive buffer (choice)					
12 13	IN	recvcount	number of elements received from any process (non-negative integer)					
14	IN	recvtype	data type of receive buffer elements (handle)					
15 16	IN	comm	communicator (handle)					
ticket 140. $\frac{17}{18}$ 19 ticket 229.1. 20	int MPI_		id* sendbuf, int sendcount, MPI_Datatype sendtype, if, int recvcount, MPI_Datatype recvtype, m)					
ticket-248T. $^{21}_{22}$	MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) BIND(C)							
23 24	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf							
24	TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcount, recvcount							
26	INTEGER, INTENT(IN) :: Sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
27								
28								
29	MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,							
30	COMM, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>							
31 32								
33	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR {void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const							
34								
35	<pre>MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype) const = 0(binding deprecated, see Section 15.2) }</pre>							
36								
37								
38	MDI	ALLTOALL is an ext	tension of MPL ALLCATHER to the case where each process					
39 40	MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each proce sends distinct data to each of the receivers. The j-th block sent from process i is received							
40			the i-th block of recvbuf.					
42		-	ated with sendcount, sendtype, at a process must be equal to					
43	the type :	signature associated	with recvcount, recvtype at any other process. This implies					
44			nust be equal to the amount of data received, pairwise between					
45		-	ual, however, the type maps may be different.					
46 47		mm is an intracommu ess (itself included) v	inicator, the outcome is as if each process executed a send to with a call to.					
47	-	· · · · · · · · · · · · · · · · · · ·						
	$\texttt{MPI}_\texttt{Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i},),$							

Unofficial Draft for Comment Only

and a receive from every other process with a call to,

```
MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...).
```

All arguments on all processes are significant. The argument **comm** must have identical values on all processes.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcount and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by recvcount and recvtype.

Rationale. For large MPI_ALLTOALL instances, allocating both send and receive buffers may consume too much memory. The "in place" option effectively halves the application memory consumption and is useful in situations where the data to be sent will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in parallel Fast Fourier Transforms). (*End of rationale.*)

Advice to implementors. Users may opt to use the "in place" option in order to conserve memory. Quality MPI implementations should thus strive to minimize system buffering. (End of advice to implementors.)

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Advice to users. When a complete exchange is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying sendcount = 0 in the reverse direction.

(End of advice to users.)

 $\overline{7}$

1 2	MPI_ALLT	FOALLV(sendbuf, sendcoun recvtype, comm)	nts, sdispls, sendtype, recvbuf, recvcounts, rdispls,				
3	IN	starting address of send buffer (choice)					
4 5 6	IN	sendcounts	non-negative integer array (of length group size) speci- fying the number of elements to send to each [processor]rank	ticket0.			
7 8 9	IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	ticket109.			
10	IN	sendtype	data type of send buffer elements (handle)				
11 12	OUT	recvbuf	address of receive buffer (choice)				
13	IN	recvcounts	non-negative integer array (of length group size) spec-				
14 15			ifying the number of elements that can be received from each [processor]rank	ticket0.			
16	IN	rdispls	integer array (of length group size). Entry i specifies	tione to.			
17 18		Tuispis	the displacement (relative to recvbuf) at which to place the incoming data from process i	ticket109.			
19 20	IN	recvtype	data type of receive buffer elements (handle)				
20	IN	comm	communicator (handle)				
22		••••					
$\begin{array}{c} {\rm ticket140.} & {}^{23} \\ {\rm ticket140.} & {}^{24} \\ {\rm ticket140.} & {}^{25} \\ {\rm ticket140.} & {}^{26} \\ {\rm ticket229.1.} & {}^{28} \\ {\rm ticket-248T.} & {}^{29} \\ & {}^{30} \\ & {}^{31} \\ & {}^{32} \\ & {}^{33} \\ & {}^{34} \\ & {}^{35} \\ & {}^{36} \\ & {}^{37} \\ & {}^{38} \\ & {}^{39} \\ & {}^{40} \\ & {}^{41} \\ & {}^{42} \end{array}$	<pre>int sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm) MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() i: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, IERROR</type></pre>						
42 43 44 45 46 47 48	<pre>{void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],</pre>						

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtype at process i must be equal to the type signature associated with recvcounts[i], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

```
MPI_Send(sendbuf + sdispls[i] · extent(sendtype), sendcounts[i], sendtype, i, ...),
```

and received a message from every other process with a call to

MPI_Recv(recvbuf + rdispls[i] · extent(recvtype), recvcounts[i], recvtype, i, ...).

All arguments on all processes are significant. The argument **comm** must have identical values on all processes.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts array and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.

Advice to users. Specifying the "in place" option (which must be given on all processes) implies that the same amount and type of data is sent and received between any two processes in the group of the communicator. Different pairs of processes can exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype on process i match recvcounts[i] and recvtype on process j. This symmetric exchange can be useful in applications where the data to be sent will not be used by the sending process after the MPI_ALLTOALLV exchange. (*End of advice to users.*)

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying **n** independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (*End of advice to implementors.*)

Unofficial Draft for Comment Only

 $\mathbf{5}$

 $\overline{7}$

 31

	1 2	MPI_ALL	FOALLW(sendbuf, recvtypes, co		sdispls,	sendtypes,	recvbuf,	recvcount	s, rdispls,	
	3	IN sendbuf IN sendcounts			starting address of send buffer (choice)					
ticket0.	4 5 6					ative integer a e number of el			/ -	ank
	7 8 9 10	IN	sdispls		the disp which to	array (of leng lacement in take the ou of integers)	bytes (rela	ative to sen	dbuf) from	
	11 12 13 14	IN	sendtypes			datatypes (the type of es)	-	,		
	14	OUT	recvbuf		address	of receive but	ffer (choice	e)		
ticket0.	16 17 18	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each [processor]rank						
	19 20 21 22	IN	rdispls	the displ to place	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)					
	23 24 25 26	IN	recvtypes			datatypes (the type of c andles)	-	- ,		
	27	IN	comm		commun	icator (hand	le)			
ticket140. ticket140. ticket140. ticket140. ticket140. ticket140. ticket140.	30 31 32 33 34 35	<pre>int MPI_Alltoallw(const void* sendbuf, const int sendcounts[], const</pre>								

Unofficial Draft for Comment Only

MPI_ALLTOALLW is the most general form of complete exchange. Like MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows separate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with

```
MPI_Send(sendbuf + sdispls[i], sendcounts[i], sendtypes[i], i, ...),
```

and received a message from every other process with a call to

```
MPI_Recv(recvbuf + rdispls[i], recvcounts[i], recvtypes[i], i, ...).
```

All arguments on all processes are significant. The argument **comm** must describe the same communicator on all processes.

Like for MPI_ALLTOALLV, the "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the received and receives arrays, and is taken from the locations of the receive buffer specified by rdispls.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have sendcounts[i] = 0, this achieves an MPI_SCATTERW function. (*End of rationale.*)

5.9 Global Reduction Operations

The functions in this section perform a global reduce operation (for example sum, maximum, ⁴⁴ and logical and) across all members of a group. The reduction operation can be either one of ⁴⁵ a predefined list of operations, or a user-defined operation. The global reduction functions ⁴⁶ come in several flavors: a reduce that returns the result of the reduction to one member of a ⁴⁷ group, an all-reduce that returns this result to all members of a group, and two scan (parallel ⁴⁸

Unofficial Draft for Comment Only

 24

```
1
                 prefix) operations. In addition, a reduce-scatter operation combines the functionality of a
            \mathbf{2}
                  reduce and of a scatter operation.
            3
            4
                 5.9.1
                         Reduce
            5
            6
            \overline{7}
                  MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
            8
                   IN
                              sendbuf
                                                           address of send buffer (choice)
            9
            10
                    OUT
                              recvbuf
                                                           address of receive buffer (choice, significant only at
            11
                                                           root)
            12
                   IN
                              count
                                                           number of elements in send buffer (non-negative inte-
            13
                                                           ger)
            14
                                                           data type of elements of send buffer (handle)
                   IN
                              datatype
            15
            16
                   IN
                                                           reduce operation (handle)
                              ор
            17
                   IN
                                                           rank of root process (integer)
                              root
            18
                   IN
                                                           communicator (handle)
                              comm
            19
            20
  ticket140. 21
                  int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
                                 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
            22
ticket-248T. 23
                  MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
            24
                                 BIND(C)
            25
                      TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
            26
                      TYPE(*), DIMENSION(..) :: recvbuf
            27
                      INTEGER, INTENT(IN) :: count, root
            28
                      TYPE(MPI_Datatype), INTENT(IN) :: datatype
            29
                      TYPE(MPI_Op), INTENT(IN) :: op
            30
                      TYPE(MPI_Comm), INTENT(IN) :: comm
            31
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            32
            33
                 MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
            34
                      <type> SENDBUF(*), RECVBUF(*)
                      INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
            35
            36
                  {void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
            37
                                 const MPI::Datatype& datatype, const MPI::Op& op, int root)
            38
                                 const = 0 (binding deprecated, see Section 15.2) }
            39
            40
                      If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the
            ^{41}
                  input buffer of each process in the group, using the operation op, and returns the combined
            42
                  value in the output buffer of the process with rank root. The input buffer is defined by
            43
                  the arguments sendbuf, count and datatype; the output buffer is defined by the arguments
            44
                  recvbuf, count and datatype; both have the same number of elements, with the same type.
            45
                  The routine is called by all group members using the same arguments for count, datatype, op,
    ticket0.<sup>46</sup>
                  root and comm. Thus, all processes provide input buffers [and output buffers] of the same
    ticket
0. ^{\rm 47}
                  length, with elements of the same type.] as the output buffer at the root. Each process
            48
                  can provide one element, or a sequence of elements, in which case the combine operation
```

CHAPTER 5. COLLECTIVE COMMUNICATION

is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)).

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 5.9.5.

The operation **op** is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of [processors]ranks. (End of advice to implementors.)

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 5.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI_GATHER), applying the reduction operation in the desired order (e.g., with MPI_REDUCE_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI_BCAST). (End of advice to users.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. It is safest to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8

9

10 11

12

13

14

15

16

17

18

19

21 22

23

 24

25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40

41 42

43

44

45

46

47 48

²⁰ ticket0.

1 The "in place" option for intracommunicators is specified by passing the value $\mathbf{2}$ MPI_IN_PLACE to the argument sendbuf at the root. In such a case, the input data is taken 3 at the root from the receive buffer, where it will be replaced by the output data. 4 If comm is an intercommunicator, then the call involves all processes in the intercom- $\mathbf{5}$ municator, but with one group (group A) defining the root process. All processes in the 6 other group (group B) pass the same value in argument root, which is the rank of the root 7in group A. The root passes the value MPI_ROOT in root. All other processes in group A 8 pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group 9 B and only receive buffer arguments are significant at the root. 10 11**Predefined Reduction Operations** 5.9.2 12The following predefined operations are supplied for MPI_REDUCE and related functions 13 MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, ticket0. 14 MPI_SCAN, [and] MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and ticket0. 15 ticket0. 16 MPI_REDUCE_LOCAL. These operations are invoked by placing the following in op. 1718 Meaning Name 1920MPI_MAX maximum 21MPI_MIN minimum 22 MPI_SUM sum 23MPI_PROD product 24 MPI_LAND logical and 25MPI_BAND bit-wise and 26logical or MPI_LOR 27MPI_BOR bit-wise or 28logical exclusive or (xor) MPI_LXOR 29bit-wise exclusive or (xor) MPI_BXOR 30 max value and location MPI_MAXLOC 31min value and location MPI_MINLOC 32 The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Sec-33 tion 5.9.4. For the other predefined operations, we enumerate below the allowed combi-34 nations of op and datatype arguments. First, define groups of MPI basic datatypes in the 35 following way. 36 37 38 MPI_INT, MPI_LONG, MPI_SHORT, C integer: 39 MPI_UNSIGNED_SHORT, MPI_UNSIGNED, 40MPI_UNSIGNED_LONG. 41 MPI_LONG_LONG_INT, 42MPI_LONG_LONG (as synonym), 43 MPI_UNSIGNED_LONG_LONG, 44MPI_SIGNED_CHAR, 45MPI_UNSIGNED_CHAR, 46MPI_INT8_T, MPI_INT16_T,

Unofficial Draft for Comment Only

MPI_INT32_T, MPI_INT64_T,

MPI_UINT8_T, MPI_UINT16_T,

47

		MPI_UINT32_T, MPI_UINT64_T	1		
ticket 265.	Fortran integer:	MPI_INTEGER, MPI_AINT, MPI_COUNT,	2		
		MPI_OFFSET,	3		
		and handles returned from	4		
		MPI_TYPE_CREATE_F90_INTEGER,	5		
		and if available: MPI_INTEGER1,	6		
		MPI_INTEGER2, MPI_INTEGER4,	7		
		MPI_INTEGER8, MPI_INTEGER16	8		
	Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	9		
		MPI_DOUBLE_PRECISION	10		
		MPI_LONG_DOUBLE	11		
		and handles returned from	12		
		MPI_TYPE_CREATE_F90_REAL,	13		
		and if available: MPI_REAL2,	14		
		MPI_REAL4, MPI_REAL8, MPI_REAL16	15		
	Logical:	MPI_LOGICAL, MPI_C_BOOL	16		
	Complex:	MPI_COMPLEX,			
		MPI_C_FLOAT_COMPLEX,	17		
		MPI_C_DOUBLE_COMPLEX,	18		
		MPI_C_LONG_DOUBLE_COMPLEX,	19		
		and handles returned from	20		
		MPI_TYPE_CREATE_F90_COMPLEX,	21		
		and if available: MPI_DOUBLE_COMPLEX,	22		
		MPI_COMPLEX4, MPI_COMPLEX8,	23		
		MPI_COMPLEX16, MPI_COMPLEX32	24		
	Byte:	MPI_BYTE	25		
	Now, the valid datatypes for each option [is] are specified below.				
	Now, the valid datatypes for each of	option isjare specified below.	$_{27}$ ticket0.		
			28		
	On	Allowed Types	29		
	Op	Anowed Types	30		
	MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point	31		
	MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex	32		
	MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	33		
	MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte	34		
		C integer, Fortrait integer, Dyte	35		
	The following examples use intracommunicators.				
			37		
	Example 5.15		38		
		product of two vectors that are distributed across a	39		
	group of processes and returns the answ	ver at node zero.	40		
			41		
			42		
			43		
			44		
			45		
			46		
			40		
			48		

```
1
     SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
\mathbf{2}
     REAL a(m), b(m)
                           ! local slice of array
3
     REAL c
                              ! result (at node zero)
4
     REAL sum
\mathbf{5}
     INTEGER m, comm, i, ierr
6
7
     ! local sum
8
     sum = 0.0
9
     DO i = 1, m
10
        sum = sum + a(i)*b(i)
^{11}
     END DO
12
13
     ! global sum
14
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
15
     RETURN
16
     END
17
18
     Example 5.16
19
         A routine that computes the product of a vector and an array that are distributed
20
     across a group of processes and returns the answer at node zero.
21
22
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
23
     REAL a(m), b(m,n)
                          ! local slice of array
^{24}
     REAL c(n)
                             ! result
25
     REAL sum(n)
26
     INTEGER n, comm, i, j, ierr
27
28
     ! local sum
29
     DO j= 1, n
30
       sum(j) = 0.0
^{31}
       D0 i = 1, m
32
         sum(j) = sum(j) + a(i)*b(i,j)
33
       END DO
34
     END DO
35
36
     ! global sum
37
     CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
38
39
     ! return result at node zero (and garbage at the other nodes)
40
     RETURN
41
     END
42
43
     5.9.3
            Signed Characters and Reductions
44
45
     The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction opera-
46
     tions. MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER (which represent printable charac-
```

ters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR,

MPI_WCHAR, and MPI_CHARACTER will be translated so as to preserve the printable

47

character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

5.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i,j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

Both operations are associative and commutative. Note that if MPI_MAXLOC is applied to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$, then the value returned is (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. Thus, if each process supplies a value and its rank within the group, then a reduce operation with op = MPI_MAXLOC will return the maximum value and the rank of the first process with that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1	according to the first component of each pair, and ties are resolved according to the second			
2	component.			
3	The reduce operation is defined to operate on arguments that consist of a pair: value			
4		types are provided to describe the pair. The potentially		
5		s is a problem in Fortran. The problem is circumvented,		
6	, , , , , , , , , , , , , , , , , , , ,	vided type consist of a pair of the same type as value,		
7	and coercing the index to this type	e also. In C, the MPI-provided pair type has distinct		
8	types and the index is an int.			
9	In order to use MPI_MINLOC and	d MPI_MAXLOC in a reduce operation, one must provide		
10	a datatype argument that represent	ts a pair (value and index). MPI provides nine such		
11	predefined datatypes. The operation	ons MPI_MAXLOC and MPI_MINLOC can be used with		
12	each of the following datatypes.			
13				
14	Fortran:			
15	Name	Description		
16	MPI_2REAL	pair of REALs		
17	MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables		
18	MPI_2INTEGER	pair of INTEGERs		
19				
20				
21	C:			
22	Name	Description		
23	MPI_FLOAT_INT	float and int		
24	MPI_DOUBLE_INT	double and int		
25	MPI_LONG_INT	long and int		
26	MPI_2INT	pair of int		
27		short and int		
28	MPI_LONG_DOUBLE_INT	long double and int		
29		-		
30	The datatype MPI_2REAL is as	if defined by the following (see Section 4.1).		
31	MDT TYPE CONTIGUOUS (O MDT DEA			
32	MPI_TYPE_CONTIGUOUS(2, MPI_REA	L, MPI_ZREAL)		
33	Similar statements apply for ME	PI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.		
34	** *	s as if defined by the following sequence of instructions.		
35	The datatype MFI_FLOAT_INT is	s as if defined by the following sequence of first actions.		
36	type[0] = MPI_FLOAT			
	type[1] = MPI_INT			
37	disp[0] = 0			
38	disp[1] = sizeof(float)			
39	block[0] = 1			
40	block[1] = 1			
41	MPI_TYPE_CREATE_STRUCT(2, bloc	k disp type MPI FLOAT INT)		
42		a, arop, oypo, in fridoni-ini,		
43	Similar statements apply for MPI_LC	DNG_INT and MPI_DOUBLE_INT.		
44	The following examples use intr	racommunicators.		
45				
46	Example 5.17			
47	Each process has an array of 30 doubles, in C. For each of the 30 locations, compute			

Each process has an array of 30 doubles, in C. For each of the 30 locations, compute
 the value and rank of the process containing the largest value.

```
1
    . . .
                                                                                        \mathbf{2}
    /* each process has an array of 30 double: ain[30]
                                                                                        3
     */
                                                                                        4
    double ain[30], aout[30];
    int ind[30];
                                                                                        5
                                                                                        6
    struct {
                                                                                        7
        double val;
                                                                                        8
              rank;
        int
    } in[30], out[30];
                                                                                       9
                                                                                       10
    int i, myrank, root;
                                                                                       11
    MPI_Comm_rank(comm, &myrank);
                                                                                       12
    for (i=0; i<30; ++i) {
                                                                                       13
                                                                                       14
        in[i].val = ain[i];
                                                                                       15
        in[i].rank = myrank;
                                                                                       16
    }
                                                                                       17
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                       18
    /* At this point, the answer resides on process root
                                                                                       19
     */
                                                                                       20
    if (myrank == root) {
                                                                                       21
        /* read ranks out
         */
                                                                                       22
        for (i=0; i<30; ++i) {</pre>
                                                                                       23
                                                                                       ^{24}
             aout[i] = out[i].val;
                                                                                       25
             ind[i] = out[i].rank;
                                                                                       26
        }
    }
                                                                                       27
                                                                                       28
                                                                                       29
Example 5.18
                                                                                       30
   Same example, in Fortran.
                                                                                       ^{31}
                                                                                       32
                                                                                       33
    ! each process has an array of 30 double: ain(30)
                                                                                       34
                                                                                       35
    DOUBLE PRECISION ain(30), aout(30)
                                                                                       36
    INTEGER ind(30)
                                                                                       37
    DOUBLE PRECISION in(2,30), out(2,30)
                                                                                       38
    INTEGER i, myrank, root, ierr
                                                                                       39
                                                                                       40
    CALL MPI_COMM_RANK(comm, myrank, ierr)
                                                                                       41
    DO I=1, 30
                                                                                       42
        in(1,i) = ain(i)
                                                                                       43
        in(2,i) = myrank ! myrank is coerced to a double
                                                                                       44
    END DO
                                                                                       45
                                                                                       46
    CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                                                       47
                                                                     comm, ierr)
                                                                                       48
```

```
1
          ! At this point, the answer resides on process root
\mathbf{2}
3
         IF (myrank .EQ. root) THEN
4
              ! read ranks out
5
              DO I= 1, 30
6
                   aout(i) = out(1,i)
7
                   ind(i) = out(2,i) ! rank is coerced back to an integer
8
              END DO
9
         END IF
10
11
     Example 5.19
12
         Each process has a non-empty array of values. Find the minimum global value, the
13
     rank of the process that holds it and its index on this process.
14
15
     #define LEN
                      1000
16
17
     float val[LEN];
                               /* local array of values */
18
                               /* local number of values */
     int count;
19
     int myrank, minrank, minindex;
20
     float minval;
21
22
     struct {
23
         float value;
^{24}
         int
                index;
25
     } in, out;
26
27
         /* local minloc */
28
     in.value = val[0];
     in.index = 0;
29
30
     for (i=1; i < count; i++)</pre>
^{31}
          if (in.value > val[i]) {
32
              in.value = val[i];
33
              in.index = i;
34
         }
35
36
         /* global minloc */
37
     MPI_Comm_rank(comm, &myrank);
38
     in.index = myrank*LEN + in.index;
39
     MPI_Reduce( &in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
40
          /* At this point, the answer resides on process root
41
           */
42
     if (myrank == root) {
43
         /* read answer out
44
           */
45
         minval = out.value;
46
         minrank = out.index / LEN;
47
         minindex = out.index % LEN;
48
     }
```

 Rationale.
 The definition of MPI_MINLOC and MPI_MAXLOC given here has the
 1

 advantage that it does not require any special-case handling of these two operations:
 2

 they are handled like any other reduce operation. A programmer can provide his or
 3

 her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage
 4

 is that values and indices have to be first interleaved, and that indices and values have
 5

 to be coerced to the same type, in Fortran. (End of rationale.)
 6

5.9.5 User-Defined Reduction Operations

			10
MPI_OP	_CREATE([function]use	er_fn, commute, op)	$^{11}_{12}$ ticket252-W.
IN	[ticket252-W.] <mark>[funct</mark>	tion]user_fn user defined function (function)	13
IN	commute	true if commutative; false otherwise.	14
OUT	ор	operation (handle)	15 16
int MPI	_Op_create(MPI_User MPI_Op* op)	_function* [function]user_fn, int commute,	¹⁷ ₁₈ ticket252-W.
PRO	create(user_fn, com CEDURE(MPI_User_fun		¹⁹ ticket-248T.
	ICAL, INTENT(IN) ::		22 23
	E(MPI_Op), INTENT(O	-	23
TNT	EGER, OPTIONAL, INT	ENT(OUT) :: ierror	25
	CREATE([FUNCTION]U ERNAL [FUNCTION]USE	I <mark>SER_FN</mark> , COMMUTE, OP, IERROR) R_FN	$_{26} ext{ ticket 252-W} = \frac{1}{27} ext{ ticket 252-W} = \frac$
LOG	ICAL COMMUTE		28
INT	EGER OP, IERROR		29
$\{void M\}$	-	<pre>Ser_function* [function]user_fn, (binding deprecated, see Section 15.2) }</pre>	$^{30}_{_{31}}$ ticket252-W.
			32
		a user-defined reduction operation to an	33
•	-	ily be used in MPI_REDUCE, MPI_ALLREDUCE,	$_{34}$ ticket $0.$
		CK, MPI_REDUCE_SCATTER, MPI_SCAN, [$_{35}$ ticket and .
		ng variants of those (see Section 5.12), and er-defined operation is assumed to be associative. If commute	$_{36}$ ticket0.
		hould be both commutative and associative. If commute	37
	=	berands is fixed and is defined to be in ascending, process	38 39
		cess zero. The order of evaluation can be changed, talking	40
	, , , , ,	of the operation. If $commute = true$ then the order of	41
evaluatio	on can be changed, tak	ing advantage of commutativity and associativity.	42
The	argument [function]use	er_fn is the user-defined function, which must have the fol-	$_{43}$ ticket252-W.
0	0	noutvec, len and datatype.	44
		he function is the following.	45
typedef		tion(void* invec, void* inoutvec, int *len,	46
	MPI_Datatype	<pre>*datatype);</pre>	47
			48

8

ticket230-B. ¹ ticket252-W. ² ticket230-B. ³ ticket-248T. 4 ⁵ 6 7 8	ABSTRACT INTERFACE SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: invec, inoutvec INTEGER :: len
ticket252-W. $\frac{10}{9}$ ticket252-W. $\frac{10}{12}$	<pre><type> INVEC(LEN), INOUTVEC(LEN)</type></pre>
15 13 14 18	The C++ declaration of the user-defined function appears below. {typedef void MPI::User_function(const void* invec, void* inoutvec, int len, const Datatype& datatype); (binding deprecated, see Section 15.2)}
16 17 18 19 20 21 22 22 22 22 23 24	The datatype argument is a handle to the data type that was passed into the call to MPI_REDUCE. The user reduce function should be written such that the following holds: Let u[0],, u[len-1] be the len elements in the communication buffer described by the arguments invec, len and datatype when the function is invoked; let v[0],, v[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let w[0],, w[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let w[0],, w[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function returns; then w[i] = u[i] \circ v[i], for i=0,, len-1, where \circ is the reduce operation that the function
22 26 ticket252-W. 27 28 26 30 31 31	Informally, we can think of invec and inoutvec as arrays of len elements that [function]user_fn is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: i.e., the function returns in inoutvec[i] the value invec[i] \circ inoutvec[i], for i = 0,, count - 1, where \circ is the combining operation computed by the function
32 33 34 35	Rationale. The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility
31 38 39	By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different
4(4) 42 42 43 44	General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies. No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of on amon
41 40 41 41	Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation according to the types of the operands. The user-defined reduce function

cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

Advice to implementors. We outline below a naive and inefficient implementation of MPI_REDUCE not supporting the "in place" option.

```
MPI_Comm_size(comm, &groupsize);
                                                                           17
MPI_Comm_rank(comm, &rank);
                                                                           18
if (rank > 0) {
                                                                           19
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
                                                                          20
    User_reduce(tempbuf, sendbuf, count, datatype);
                                                                          21
}
                                                                          22
if (rank < groupsize-1) {</pre>
                                                                          23
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
                                                                           ^{24}
}
                                                                           25
/* answer now resides in process groupsize-1 ... now send to root
                                                                           26
 */
                                                                          27
if (rank == root) {
                                                                          28
    MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
                                                                          29
}
                                                                          30
if (rank == groupsize-1) {
                                                                           31
    MPI_Send(sendbuf, count, datatype, root, ...);
                                                                           32
}
                                                                           33
if (rank == root) {
                                                                          34
    MPI_Wait(&req, &status);
                                                                          35
}
                                                                          36
```

The reduction computation proceeds, sequentially, from process 0 to process groupsize-1. This order is chosen so as to respect the order of a possibly noncommutative operator defined by the function User_reduce(). A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size len <count.

The predefined reduce operations can be implemented as a library of user-defined ⁴⁶ operations. However, better performance might be achieved if MPI_REDUCE handles ⁴⁷ these functions as a special case. (*End of advice to implementors.*) ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

37

38

39

40

41

42

43

44

```
1
                 MPI_OP_FREE(op)
            \mathbf{2}
                   INOUT
                                                         operation (handle)
                             ор
            3
            4
                 int MPI_Op_free(MPI_Op *op)
            5
ticket-248T.
            6
                 MPI_Op_free(op, ierror) BIND(C)
            7
                      TYPE(MPI_Op), INTENT(INOUT) :: op
            8
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            9
                 MPI_OP_FREE(OP, IERROR)
            10
                      INTEGER OP, IERROR
           11
           12
                 {void MPI::Op::Free()(binding deprecated, see Section 15.2)}
           13
                      Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
           14
           15
           16
                 Example of User-defined Reduce
            17
                 It is time for an example of user-defined reduction. The example in this section uses an
            18
                 intracommunicator.
            19
           20
                 Example 5.20 Compute the product of an array of complex numbers, in C.
           21
                 typedef struct {
           22
                      double real, imag;
           23
                 } Complex;
           ^{24}
           25
           26
                 /* the user-defined function
                  */
           27
                 void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
           28
                 {
           29
                      int i;
           30
                      Complex c;
           ^{31}
                      Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
           32
           33
           34
                      for (i=0; i< *len; ++i) {</pre>
                          c.real = inout->real*in->real -
           35
                                       inout->imag*in->imag;
           36
                          c.imag = inout->real*in->imag +
           37
                                       inout->imag*in->real;
           38
                          *inout = c;
           39
                          in++; inout++;
            40
                      }
           41
                 }
           42
           43
                 /* and, to call it...
           44
                  */
           45
           46
                 . . .
           47
                      /* each process has an array of 100 Complexes
           48
```

```
1
     */
                                                                                      \mathbf{2}
    Complex a[100], answer[100];
                                                                                      3
    MPI_Op myOp;
    MPI_Datatype ctype;
                                                                                      4
                                                                                      5
    /* explain to MPI how type Complex is defined
                                                                                      6
                                                                                      7
     */
                                                                                      8
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
    MPI_Type_commit(&ctype);
                                                                                      9
                                                                                      10
    /* create the complex-product user-op
                                                                                      11
     */
    MPI_Op_create( myProd, 1, &myOp );
                                                                                      12
                                                                                      13
    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
                                                                                     14
                                                                                      15
                                                                                      16
    /* At this point, the answer, which consists of 100 Complexes,
                                                                                      17
     * resides on process root
                                                                                      18
     */
                                                                                      19
                                                                                     20
                                                                                       ticket229.2.
Example 5.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
                                                                                      21
                                                                                        ticket230-B.
                                                                                     22
 subroutine my_user_function( invec, inoutvec, len, type )
                                                                   bind(c)
                                                                                     23
    use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
                                                                                     ^{24}
    type(c_ptr), value :: invec, inoutvec
                                                                                     25
    integer :: len
                                                                                      26
    type(MPI_Datatype) :: type
                                                                                     27
    real, pointer :: invec_r(:), inoutvec_r(:)
                                                                                     28
    if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
                                                                                     29
       call c_f_pointer(invec, invec_r, (/ len /) )
                                                                                     30
       call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
```

end if end subroutine

inoutvec_r = invec_r + inoutvec_r

5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results. 31

32

33

34 35

36 37

38

39

1 MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm) 2 IN sendbuf starting address of send buffer (choice) 3 OUT recvbuf starting address of receive buffer (choice) 4 5IN count number of elements in send buffer (non-negative inte-6 ger) 7 IN datatype data type of elements of send buffer (handle) 8 IN ор operation (handle) 9 10 IN comm communicator (handle) 11 12ticket140. int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, 13 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) ticket-248T. 14 MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C) 15TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 16TYPE(*), DIMENSION(..) :: recvbuf 17 INTEGER, INTENT(IN) :: count 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19TYPE(MPI_Op), INTENT(IN) :: op 20TYPE(MPI_Comm), INTENT(IN) :: comm 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 24<type> SENDBUF(*), RECVBUF(*) 25INTEGER COUNT, DATATYPE, OP, COMM, IERROR 26{void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, 27const MPI::Datatype& datatype, const MPI::Op& op) 28const = O(binding deprecated, see Section 15.2)29 30 If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as 31 MPI_REDUCE except that the result appears in the receive buffer of all the group members. 32 33 Advice to implementors. The all-reduce operations can be implemented as a re-34 duce, followed by a broadcast. However, a direct implementation can lead to better 35 performance. (End of advice to implementors.) 36 37 The "in place" option for intracommunicators is specified by passing the value 38MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is 39 taken at each process from the receive buffer, where it will be replaced by the output data. 40 If comm is an intercommunicator, then the result of the reduction of the data provided 41 by processes in group A is stored at each process in group B, and vice versa. Both groups 42should provide **count** and **datatype** arguments that specify the same type signature. 43 The following example uses an intracommunicator. 44 45Example 5.22 46A routine that computes the product of a vector and an array that are distributed 47across a group of processes and returns the answer at all nodes (see also Example 5.16).

```
1
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
                                                                                             \mathbf{2}
                        ! local slice of array
REAL a(m), b(m,n)
                                                                                             3
REAL c(n)
                         ! result
REAL sum(n)
                                                                                             4
INTEGER n, comm, i, j, ierr
                                                                                             5
                                                                                             6
                                                                                             7
! local sum
                                                                                             8
DO j= 1, n
  sum(j) = 0.0
                                                                                             9
                                                                                             10
  DO i = 1, m
                                                                                             11
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
                                                                                             12
END DO
                                                                                             13
                                                                                             14
                                                                                             15
! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
                                                                                             16
                                                                                             17
                                                                                             18
! return result at all nodes
                                                                                             19
RETURN
                                                                                             20
END
                                                                                             21
                                                                                             ^{22} ticket0.
5.9.7
       Process-II Local [r] Reduction
                                                                                             ^{23} ticket0.
The functions in this section are of importance to library implementors who may want to
                                                                                             ^{24}
implement special reduction patterns that are otherwise not easily covered by the standard
                                                                                             25
MPI operations.
                                                                                             26
    The following function applies a reduction operator to local arguments.
                                                                                             27
                                                                                             28
                                                                                             29
MPI_REDUCE_LOCAL( inbuf, inoutbuf, count, datatype, op)
                                                                                             30
  IN
            inbuf
                                         input buffer (choice)
                                                                                             31
                                                                                             32
  INOUT
            inoutbuf
                                         combined input and output buffer (choice)
                                                                                             33
  IN
                                         number of elements in inbuf and inoutbuf buffers (non-
            count
                                                                                             34
                                         negative integer)
                                                                                             35
  IN
            datatype
                                         data type of elements of inbuf and inoutbuf buffers
                                                                                             36
                                         (handle)
                                                                                             37
                                                                                             38
  IN
                                         operation (handle)
            ор
                                                                                             39
                                                                                             ^{40} ticket 140.
int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,
                                                                                             41
               MPI_Datatype datatype, MPI_Op op)
                                                                                             ^{42} ticket-248T.
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C)
                                                                                             43
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
                                                                                             44
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                             45
    INTEGER, INTENT(IN) :: count
                                                                                             46
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                             47
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                             48
```

```
1
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            2
ticket250-V.
                 MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
                      <type> INBUF(*), INOUTBUF(*)
            4
                      INTEGER COUNT, DATATYPE, OP, IERROR
            5
            6
                 {void MPI::Op::Reduce_local(const void* inbuf, void* inoutbuf, int count,
            7
                                 const MPI::Datatype& datatype) const/binding deprecated, see
            8
                                 Section 15.2 }
            9
                     The function applies the operation given by op element-wise to the elements of inbuf
            10
                 and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined
            11
                 operations in Section 5.9.5. Both inbuf and inoutbuf (input as well as result) have the
           12
                 same number of elements given by count and the same datatype given by datatype. The
           13
                 MPI_IN_PLACE option is not allowed.
           14
                     Reduction operations can be queried for their commutativity.
           15
            16
            17
                 MPI_OP_COMMUTATIVE( op, commute)
            18
                   IN
                                                         operation (handle)
                             op
           19
                   OUT
           20
                             commute
                                                         true if op is commutative, false otherwise (logical)
           21
           22
                 int MPI_Op_commutative(MPI_Op op, int *commute)
ticket-248T. 23
                 MPI_Op_commutative(op, commute, ierror) BIND(C)
           24
                      TYPE(MPI_Op), INTENT(IN) :: op
           25
                      LOGICAL, INTENT(OUT) :: commute
            26
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           27
           28
                 MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
           29
                      LOGICAL COMMUTE
           30
                      INTEGER OP, IERROR
           ^{31}
                 {bool MPI::Op::Is_commutative() const(binding deprecated, see Section 15.2) }
           32
           33
           34
                         Reduce-Scatter
                 5.10
           35
           36
                 MPI includes variants of the reduce operations where the result is scattered to all processes
           37
                 in a group on return. One variant scatters equal-sized blocks to all processes, while another
           38
                 variant scatters blocks that may vary in size for each process.
           39
            40
           41
           42
           43
           44
           45
            46
            47
            48
```

5.10.1 MPI_REDUCE_SCATTER_BLOCK

 $\mathbf{2}$ 3 4 MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm) 5 IN sendbuf starting address of send buffer (choice) 6 OUT recvbuf starting address of receive buffer (choice) 7 8 IN recvcount element count per block (non-negative integer) 9 IN datatype data type of elements of send and receive buffers (han-10 dle) 11 IN operation (handle) op 1213 IN comm communicator (handle) 14 15int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf, ticket140. 16 int recvcount, MPI_Datatype datatype, MPI_Op op, 17 MPI_Comm comm) ¹⁸ ticket-248T. MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 19 ierror) BIND(C) 20TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 21TYPE(*), DIMENSION(...) :: recvbuf 22 INTEGER, INTENT(IN) :: recvcount 23TYPE(MPI_Datatype), INTENT(IN) :: datatype 24 TYPE(MPI_Op), INTENT(IN) :: op 25TYPE(MPI_Comm), INTENT(IN) :: comm 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728 MPI_REDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 29 IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 32 {void MPI::Comm::Reduce_scatter_block(const void* sendbuf, void* recvbuf, 33 int recvcount, const MPI::Datatype& datatype, 34 const MPI::Op& op) const = O(binding deprecated, see Section 15.2) } 35 36 If comm is an intracommunicator, MPI_REDUCE_SCATTER_BLOCK first performs a 37

If comm is an intracommunicator, MPI_REDUCE_SCATTER_BLOCK first performs a global, element-wise reduction on vectors of count = n^* recvcount elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcount, datatype, op and comm. The resulting vector is treated as n consecutive blocks of recvcount elements that are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcount, and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER_BLOCK routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to recvcount*n, followed by an MPI_SCATTER with sendcount equal to recvcount. However, a direct implementation may run faster. (End of advice to implementors.) 48

Unofficial Draft for Comment Only

1

38

39

40

41

42

43

ticket0.	$\cdot \frac{1}{2}$	The "in place" option for intracommunic [ators is specified by passing MPI_IN_PLACE in the sendbuf argument on <i>all</i> processes. In this case, the input data is taken from the receive buffer.				
	4	If comm is an intercommunicator, then the result of the reduction of the data provided				
	5	by processes in one group (group A) is scattered among processes in the other group (group				
	6	B) and vice versa. Within each group, all processes provide the same value for the recvcount				
	7	argument, and provide input vectors of $count = n^* recvcount$ elements stored in the send				
	8	buffers, where n is the size of the group. The number of elements count must be the same				
	9	for the two groups. The resulting vector from the other group is scattered in blocks of				
	10	recvcount elements among the processes in the group.				
	11	Datio	male The last restriction i	a needed as that the length of the cond buffer of		
	12 13			s needed so that the length of the send buffer of the local recvcount argument of the other group.		
	14			ded to figure out how many elements are reduced.		
	15		of rationale.)	and to figure out now many clements are reduced.		
	16		<i></i>			
	17	5.10.2 M	PI_REDUCE_SCATTER			
	18		ICE SCATTEP artends the f	unotionality of MPI PEDLICE SCATTER RIOCK		
	¹⁹ MPI_REDUCE_SCATTER extends the functionality of MPI_REDUCE_SCATTER. ²⁰ such that the scattered blocks can vary in size. Block sizes are determined by the re			-		
	20					
	array, such that the <i>i</i> -th block contains recvcounts[i] elements.					
	22	MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)				
	24	MPI_REDU	JCE_SCATTER(sendbut, recvt	out, recvcounts, datatype, op, comm)		
	25	IN	sendbuf	starting address of send buffer (choice)		
	26	OUT	recvbuf	starting address of receive buffer (choice)		
	27	IN	recvcounts	non-negative integer array (of length group size) spec-		
	28			ifying the number of elements of the result distributed		
	29 30			to each process.		
	31	IN	datatype	data type of elements of send and receive buffers (han-		
	32			dle)		
	33	IN	ор	operation (handle)		
	34	IN	comm	communicator (handle)		
	35					
ticket140. ticket140.	38	<pre>int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const</pre>				
ticket-248T.	. 40					
	40	<pre>MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror) BIND(C)</pre>				
	42	TYPE(*), DIMENSION(), INTENI	(TN) :: sendbuf		
	43		*), DIMENSION() :: re			
	44		ER, INTENT(IN) :: recvcc			
	45	TYPE(MPI_Datatype), INTENT(IN)	:: datatype		
	46		<pre>MPI_Op), INTENT(IN) :: c</pre>	·		
	47		<pre>MPI_Comm), INTENT(IN) ::</pre>			
	48	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				

int recvcounts[], const MPI::Datatype& datatype, const MPI::Op& op) const = O(binding deprecated, see Section 15.2) }

If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[i]==0 may not have allocated a receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same recvcounts argument, and provide input vectors of count = $\sum_{i=0}^{n-1} \text{recvcounts}[i]$ elements stored in the send buffers, where n is the size of the group. The resulting vector from the other group is scattered in blocks of recvcounts[i] elements among the processes in the group. The number of elements count must be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local **recvcounts** entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

 24

1	5.11	Scan				
2 3	5.11.1	5.11.1 Inclusive Scan				
4						
6	MPL S	CAN(sendbuf_recybuf	count, datatype, op, comm)			
7	IN IN	sendbuf	starting address of send buffer (choice)			
8	OUT	recvbuf	- · · · · · · · · · · · · · · · · · · ·			
9 10			starting address of receive buffer (choice)			
11	IN	count	number of elements in input buffer (non-negative in-teger)			
12 13	IN	datatype	data type of elements of input buffer (handle)			
14	IN	ор	operation (handle)			
15	IN	comm	communicator (handle)			
16						
ticket 140. $\frac{17}{18}$	int MP		<pre>* sendbuf, void* recvbuf, int count, MDL Community</pre>			
ticket-248T. ¹⁹		MP1_Dataty	pe datatype, MPI_Op op, MPI_Comm comm)			
20			if, count, datatype, op, comm, ierror) BIND(C)			
21 22		TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf				
23		INTEGER, INTENT(IN) :: count				
24		TYPE(MPI_Datatype), INTENT(IN) :: datatype				
25		TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
26 27						
28						
29		ype> SENDBUF(*), H	JF, COUNT, DATATYPE, OP, COMM, IERROR) RECVBUF(*)			
30		INTEGER COUNT, DATATYPE, OP, COMM, IERROR				
31 32	{void	<pre>{void MPI::Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,</pre>				
33	(const MPI::Datatype& datatype, const MPI::Op& op) const $(binding$				
34		deprecated, s	see Section 15.2 }			
35 36	If	comm is an intracom	umunicator, MPI_SCAN is used to perform a prefix reduction			
30		on data distributed across the group. The operation returns, in the receive buffer of the				
38	-	,	uction of the values in the send buffers of processes with ranks be of operations supported, their semantics, and the constraints			
39		,	are as for MPI_REDUCE.			
40 41			or intracommunicators is specified by passing MPI_IN_PLACE in			
41 42			his case, the input data is taken from the receive buffer, and			
43	-	d by the output data				
44	11.	is operation is invalid	l for intercommunicators.			
45						
46 47						
48						

5.11.	207				
5.11.	5.11.2 Exclusive Scan				
			2 3		
			3		
MPI_	EXSCAN(sendbuf, recvbuf,	count, datatype, op, comm)	* 5		
IN	sendbuf	starting address of send buffer (choice)	6		
OU	T recvbuf	starting address of receive buffer (choice)	7		
IN	count	number of elements in input buffer (non-negative	in- ⁸		
		teger)	9		
IN	datatype	data type of elements of input buffer (handle)	10		
IN			11 12		
	ор	operation (handle)	12		
IN	comm	intracommunicator (handle)	14		
int 1	int MPI_Exscan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) ¹⁵ ¹⁶ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁶ ¹⁷ ¹⁷ ¹⁶ ¹⁷ ¹⁶ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁶ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷ ¹⁷				
		f, count, datatype, op, comm, ierror) BIND(C)	18		
		, INTENT(IN) :: sendbuf	19 20		
	<pre>FYPE(*), DIMENSION() INTEGER, INTENT(IN) ::</pre>		20 21		
	TYPE(MPI_Datatype), IN		22		
	<pre>FYPE(MPI_Op), INTENT(II</pre>	• -	23		
	TYPE(MPI_Comm), INTENT	-	24		
	INTEGER, OPTIONAL, INT	ENT(OUT) :: ierror	25		
мрті		F, COUNT, DATATYPE, OP, COMM, IERROR)	26		
	<pre><type> SENDBUF(*), REC'</type></pre>		27		
	INTEGER COUNT, DATATYP		28		
			29 + 30		
{void		an(const void* sendbuf, void* recvbuf, int coun	. u ,		
	const MP1::Da deprecated, see	<pre>statype& datatype, const MPI::Op& op) const(binds Section 15 2) }</pre>	ang 31 32		
	ueprecureu, see	Scenon 10.2) {	33		
I	If comm is an intracommunicator, MPL EXSCAN is used to perform a prefix reduction				

If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i - 1$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

34

35

36

37

38

39

40

41

42

43

44

 $45 \\ 46$

47

48

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data. The receive buffer on rank 0 is not changed by this operation.

This operation is invalid for intercommunicators.

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining

```
1
            the local contribution. Note that for non-invertable operations such as MPI_MAX, the
\mathbf{2}
            exclusive scan cannot be computed with the inclusive scan. (End of rationale.)
3
4
      5.11.3 Example using MPI_SCAN
5
      The example in this section uses an intracommunicator.
6
7
      Example 5.23
8
           This example uses a user-defined operation to produce a segmented scan. A segmented
9
      scan takes, as input, a set of values and a set of logicals, and the logicals delineate the
10
      various segments of the scan. For example:
11
                   values
                             12
                   logicals 0
13
                   result
14
15
           The operator that produces this effect is,
16
                                        \left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\j\end{array}\right),
17
18
19
           where,
20
21
                                         w = \begin{cases} u+v & \text{if } i=j \\ v & \text{if } i\neq j \end{cases}.
22
23
           Note that this is a non-commutative operator. C code that implements it is given
24
      below.
25
26
      typedef struct {
27
           double val;
28
           int log;
29
      } SegScanPair;
30
^{31}
      /* the user-defined function
32
       */
33
      void segScan(SegScanPair *in, SegScanPair *inout, int *len,
34
                                                                    MPI_Datatype *dptr)
35
      {
36
           int i;
37
           SegScanPair c;
38
39
           for (i=0; i< *len; ++i) {</pre>
40
                if (in->log == inout->log)
41
                      c.val = in->val + inout->val;
42
                else
43
                      c.val = inout->val;
44
                c.log = inout->log;
45
                *inout = c;
46
                in++; inout++;
47
           }
```

CHAPTER 5. COLLECTIVE COMMUNICATION

208

48

}

Note that the inout argument to the user-defined function corresponds to the righthand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
int i,base;
SegScanPair
             a, answer;
MPI_Op
             myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint
             disp[2];
             blocklen[2] = \{ 1, 1\};
int
MPI_Datatype sspair;
/* explain to MPI how type SegScanPair is defined
 */
MPI_Get_address( &a, disp);
MPI_Get_address( &a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;</pre>
MPI_Type_create_struct( 2, blocklen, disp, type, &sspair );
MPI_Type_commit( &sspair );
/* create the segmented-scan user-op
 */
MPI_Op_create(segScan, 0, &myOp);
MPI_Scan( &a, &answer, 1, sspair, myOp, comm );
```

5.12Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by overlapping communication and computation, and many systems enable this. Nonblocking collective operations combine the potential benefits of nonblocking point-to-point operations, to exploit overlap and to avoid synchronization, with the optimized implementation and message scheduling provided by collective operations [30, 34]. One way of doing this would be to perform a blocking collective operation in a separate thread. An alternative mechanism that often leads to better performance (e.g., avoids context switching, scheduler overheads, and thread management) is to use nonblocking collective communication [32].

37 The nonblocking collective communication model is similar to the model used for nonblocking point-to-point communication. A nonblocking call initiates a collective operation, 38 39 which must be completed in a separate completion call. Once initiated, the operation may progress independently of any computation or other communication at participating 41 processes. In this manner, nonblocking collective operations can mitigate possible synchro-42nizing effects of collective operations by running them in the "background." In addition to enabling communication-computation overlap, nonblocking collective operations can per-43 44form collective operations on overlapping communicators, which would lead to deadlocks with blocking operations. Their semantic advantages can also be useful in combination with 45point-to-point communication.

47As in the nonblocking point-to-point case, all calls are local and return immediately, 48 irrespective of the status of other processes. The call initiates the operation, which indicates

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11 12

13

14

15

16

17

18

19

20

21

22

2324

2627

28 29

30

31

32 33

34

35

36

40

46

 25 ticket 109.

1 that the system may start to copy data out of the send buffer and into the receive buffer. $\mathbf{2}$ Once initiated, all associated send buffers and buffers associated with input arguments (such 3 as arrays of counts, displacements, or datatypes in the vector versions of the collectives) 4 should not be modified, and all associated receive buffers should not be accessed, until the $\mathbf{5}$ collective operation completes. The call returns a request handle, which must be passed to 6 a completion call.

7 All completion calls (e.g., MPI_WAIT) described in Section 3.7.3 are supported for 8 nonblocking collective operations. Similarly to the blocking case, nonblocking collective 9 operations are considered to be complete when the local part of the operation is finished, 10 i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be 11safely accessed and modified. Completion does not indicate that other processes have 12completed or even started the operation (unless otherwise implied by the description of 13the operation). Completion of a particular nonblocking collective operation also does not 14indicate completion of any other posted nonblocking collective (or send-receive) operations, 15whether they are posted before or after the completed operation.

Users should be aware that implementations are allowed, but Advice to users. not required (with exception of MPI_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation. (End of advice to users.)

Upon returning from a completion call in which a nonblocking collective operation 21completes, the MPI_ERROR field in the associated status object is set appropriately, see 22 Section 3.2.5 on page 34. The values of the MPI_SOURCE and MPI_TAG fields are undefined. 23It is valid to mix different request types (i.e., any combination of collective requests, I/O 24requests, generalized requests, or point-to-point requests) in functions that enable multiple 25completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or 26MPI_CANCEL for a request associated with a nonblocking collective operation. Nonblocking 27collective requests are not persistent.

28 29

30

 31

32

33

16

17

18

1920

> Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (End of rationale.)

34Multiple nonblocking collective operations can be outstanding on a single communi-35 cator. If the nonblocking call causes some system resource to be exhausted, then it will 36 fail and generate an MPI exception. Quality implementations of MPI should ensure that 37 this happens only in pathological cases. That is, an MPI implementation should be able to 38 support a large number of pending nonblocking operations.

39 Unlike point-to-point operations, nonblocking collective operations do not match with 40 blocking collective operations, and collective operations do not have a tag argument. All 41 processes must call collective operations (blocking and nonblocking) in the same order 42per communicator. In particular, once a process calls a collective operation, all other 43 processes in the communicator must eventually call the same collective operation, and no 44other collective operation with the same communicator in between. This is consistent with 45the ordering rules for blocking collective operations in threaded environments.

46

48

47Matching blocking and nonblocking collective operations is not allowed Rationale. because the implementation might use different communication algorithms for the two

5.12. NONBLOCKING COLLECTIVE OPERATIONS

cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

The use of tags for collective operations can prevent certain hardware optimizations. (*End of rationale.*)

Advice to users. If program semantics require matching blocking and nonblocking collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (*End of advice to users.*)

In terms of data movements, each nonblocking collective operation has the same effect as its blocking counterpart for intracommunicators and intercommunicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

Advice to implementors. Nonblocking collective operations can be implemented with local execution schedules [33] using nonblocking point-to-point communication and a reserved tag-space. (End of advice to implementors.)

5.12.1 Nonblocking Barrier Synchronization

MPI_IBARRIER	(comm	, request)	
--------------	-------	------------	--

WFI_IDA		uest)	32	
IN	comm	communicator (handle)	33	
OUT	request	communication request (handle)	34	
		- 、 /	35	
int MPT	Tharrier(MPT Co	mm comm, MPI_Request *request)	36	
1110 III 1 <u>-</u>		mm comm, in i_nequebe (iequebe)	³⁷ ticket-248T.	
	-	uest, ierror) BIND(C)	38	
	-	ENT(IN) :: comm	39	
TYPE	E(MPI_Request),	INTENT(OUT) :: request	40	
INTE	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
MPT TRAF	RIER(COMM, REQU	IFST TERROR)	42	
	GER COMM, REQUE	-	43	
1111			44 ticket272.	
[{MF	'I::Request MPI:	:Comm::Ibarrier() const = O(binding deprecated, see	45	
Section		$15.2)$ }	46	
			47	

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11 12

13

14

15

16

17

18

19

20

21

22

23

 $\frac{24}{25}$

26

27 28

29 30 31

1	1						
2	MPI_I	MPI_IBARRIER is a nonblocking version of MPI_BARRIER. By calling MPI_IBARRIER,					
3	a process notifies that it has reached the barrier. The call returns immediately, in						
4		dent of whether other processes have called MPI_IBARRIER. The usual barrier semantics are enforced at the corresponding completion operation (test or wait), which in the intra-					
5							
6		communicator case will complete only after all other processes in the communicator have called MPI_IBARRIER. In the intercommunicator case, it will complete when all processes in the remote group have called MPI_IBARRIER.					
7							
9	m the rem						
10	Advice to users. A nonblocking barrier can be used to hide latency. Moving independent						
11	dent computations between the MPI_IBARRIER and the subsequent completion c can overlap the barrier latency and therefore shorten possible waiting times. The						
12							
13			o useful when mixing collective operations and point-to-point				
14	mess	ages. (End of advic	e to users.)				
15 16	F 10.0 N	and the state of December					
17	5.12.2 N	onblocking Broadca	IST				
18							
19	MPL IRCA	ST(buffer count da	tatype, root, comm, request)				
20	INOUT	buffer	· · · · · · · · · · · · · · · · · · ·				
21			starting address of buffer (choice)				
22	IN	count	number of entries in buffer (non-negative integer)				
23 24	IN	datatype	data type of buffer (handle)				
25	IN	root	rank of broadcast root (integer)				
26	IN	comm	communicator (handle)				
27 28	OUT	request	communication request (handle)				
29							
ticket 140. $_{30}$	int MPI_I	int MPI_Ibcast(const void* buffer, int count, MPI_Datatype datatype,					
ticket- 248 T. ³¹		int root, MH	PI_Comm comm, MPI_Request *request)				
32	MPI_Ibcas	t(buffer, count,	<pre>datatype, root, comm, request, ierror) BIND(C)</pre>				
33		<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer</pre>					
34		ER, INTENT(IN) :					
35 36		TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm					
37			TENT(OUT) :: request				
38		ER, OPTIONAL, IN					
39							
40	_	ST(BUFFER, COUNT, >> BUFFER(*)	DATATYPE, ROOT, COMM, REQUEST, IERROR)				
41	J 1		PE, ROOT, COMM, REQUEST, IERROR				
ticket272. 42							
43 44	[{MPI		omm::Ibcast(void* buffer, int count,				
44 45			Datatype& datatype, int root) const = 0(binding				
46		ueprecatea, se	$e Section (15.2) \}$				
47]	11 1 .					
48	This c	all starts a nonbloc	king variant of MPI_BCAST (see Section 5.4).				

- ·			1		
Example ι	using MPI_IBCAST		2		
The exam	ple in this section use	es an intracommunicator.	3		
D 1	F 0.4		4		
Example		• • • • • • • • • • • • • • • • • • •	5		
		ts from process 0 to every process in the group, perform some	6		
computat	ion on independent da	ata, and then complete the outstanding broadcast operation.	7		
мрт	Comm comm;		8		
	array1[100], array	2[100]:	9		
	root=0;		10		
	Request req;		11		
			12		
	Ibcast(arrav1, 100	, MPI_INT, root, comm, &req);	13		
	ute(array2, 100);	,,,,,	14		
_	Wait(&req, MPI_STA	TUS_IGNORE);	15		
_	· · · · · · · · · · · · · · · · · · ·		16		
5.12.3 N	Nonblocking Gather		17		
0.12.0			18		
			19		
	THER(condbuf condec	ount, sendtype, recvbuf, recvcount, recvtype, root, comm,	20		
	request)	sunt, sendtype, recybur, recycount, recytype, root, comm,	21		
	. ,		22		
IN	sendbuf	starting address of send buffer (choice)	23		
IN	sendcount	number of elements in send buffer (non-negative inte-	24		
		$\operatorname{ger})$	25		
IN	sendtype	data type of send buffer elements (handle)	26		
			27		
OUT	recvbuf	address of receive buffer (choice, significant only at root)	28		
			29		
IN	recvcount	number of elements for any single receive (non-negative	30		
		integer, significant only at root)	31 32		
IN	recvtype	data type of recv buffer elements (significant only at	33		
		root) (handle)	34		
IN	root	rank of receiving process (integer)	35		
IN	comm	communicator (handle)	36		
			37		
OUT	request	communication request (handle)	38		
			39		
int MPI_	· · · · · · · · · · · · · · · · · · ·	* sendbuf, int sendcount, MPI_Datatype sendtype,	$_{40}$ ticket 140.		
		, int recvcount, MPI_Datatype recvtype, int root,	41		
	MPI_Comm comm	n, MPI_Request *request)	42 ticket-248T.		
MPI Igat	her(sendbuf. sendc	ount, sendtype, recvbuf, recvcount, recvtype,	43		
00		cequest, ierror) BIND(C)	44		
TYPE		, INTENT(IN), ASYNCHRONOUS :: sendbuf	45		
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf					
INTEGER, INTENT(IN) :: sendcount, recvcount, root 47					
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 48				

1 2 3	TYPE	(MPI_Comm), INTE (MPI_Request), 1 GER, OPTIONAL, 1	NTENT(OUT) :: request	
4 5 6 7 8 ticket272 . ⁹	<pre>MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR</type></pre>			
10 11 12 13 14 15 16	<pre>[{MPI::Request MPI::Comm::Igather(const void* sendbuf, int sendcount,</pre>			
17 18 19 20	MPI_IGA	ГНЕRV(sendbuf, se comm, reque	ndcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, st)	
20	IN	sendbuf	starting address of send buffer (choice)	
22 23	IN	sendcount	number of elements in send buffer (non-negative integer)	
24	IN	sendtype	data type of send buffer elements (handle)	
25 26 27	OUT	recvbuf	address of receive buffer (choice, significant only at root)	
28 29 30	IN	recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process (significant only at root)	
31 32 33 34	IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)	
35 36 37	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	
38	IN	root	rank of receiving process (integer)	
39	IN	comm	communicator (handle)	
40 41	OUT	request	communication request (handle)	
42 ticket140. 43 ticket140. 44 ticket140. 44 ticket140. 45 ticket229.1. 46	<pre>int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>			
ticket-248T. ⁴⁷ MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, di 48 recvtype, root, comm, request, ierror) BIND(C)				

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    1
                                                                                    \mathbf{2}
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    3
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                    4
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    5
                                                                                    6
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    7
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    9
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                    10
              RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                    11
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   12
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                   13
    COMM, REQUEST, IERROR
                                                                                   ^{14} ticket272.
                                                                                   15
    [{MPI::Request MPI::Comm::Igatherv(const void* sendbuf, int sendcount,
                                                                                   16
              const MPI::Datatype& sendtype, void* recvbuf,
                                                                                   17
              const int recvcounts[], const int displs[],
                                                                                   18
              const MPI::Datatype& recvtype, int root) const = 0(binding
                                                                                   19
              deprecated, see Section 15.2 }
                                                                                   20
                                                                                   21
    This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).
                                                                                   22
                                                                                   23
```

```
5.12.4 Nonblocking Scatter
```

			20
MPI_ISCA	ATTER(sendbuf, s	sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,	27
	request)		28
IN	sendbuf	address of send buffer (choice, significant only at root)	29
IN	sendcount	number of elements sent to each process (non-negative	30 31
IIN	Senacount	integer, significant only at root)	
		integer, significant only at 1000)	32
IN	sendtype	data type of send buffer elements (significant only at	33
		root) (handle)	34
OUT	recvbuf	address of receive buffer (choice)	35
	recybur		36
IN	recvcount	number of elements in receive buffer (non-negative in-	37
		$\operatorname{teger})$	38
IN	recvtype	data type of receive buffer elements (handle)	39
IN	root	rank of sending process (integer)	40
IIN	1001	Tank of sending process (integer)	41
IN	comm	communicator (handle)	42
OUT	request	communication request (handle)	43
			44
			45

Unofficial Draft for Comment Only

1	MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,					
2 3	root, comm, request, ierror) BIND(C)					
4	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf					
5	INTEGER, INTENT(IN) :: sendcount, recvcount, root					
6	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype					
7	TYPE(MPI_Comm), INTENT(IN) :: comm					
8	TYPE(MPI_Request), INTENT(OUT) :: request					
9	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
10 11	<pre>MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) </pre> <pre></pre>					
12						
13						
14						
ticket272. 15	IERF	UK				
16	<pre>[{MPI::Request MPI::Comm::Iscatter(const void* sendbuf, int sendcount,</pre>					
17 18	const MPI::Datatype& sendtype, void* recvbuf, int recvcount,					
19	<pre>const MPI::Datatype& recvtype, int root) const = 0(binding democrated acc Section 15.2)</pre>					
20	deprecated, see Section 15.2 }					
21]					
22	This	call starts a nonb	locking variant of $MPI_SCATTER$ (see Section 5.6).			
23						
24 25	MPI_ISC/	× •	sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,			
26		comm, requ	est)			
27	IN	sendbuf	address of send buffer (choice, significant only at root)			
²⁸ ticket0. ²⁹	IN	sendcounts	non-negative integer array (of length group size) speci- fying the number of elements to send to each [processor]rank			
30	IN	displs	integer array (of length group size). Entry i specifies			
31 32			the displacement (relative to sendbuf) from which to			
33			take the outgoing data to process i			
34	IN	sendtype	data type of send buffer elements (handle)			
35	OUT	recvbuf	address of receive buffer (choice)			
36 37	IN	recvcount	number of elements in receive buffer (non-negative in-			
38			teger)			
39	IN	recvtype	data type of receive buffer elements (handle)			
40	IN	root	rank of sending process (integer)			
41	IN	comm	communicator (handle)			
42 43	OUT	request	communication request (handle)			
43						
ticket140. 45 ticket140. 46 ticket140. 47 48	<pre>int MPI_Iscatterv(const void* sendbuf, const int sendcounts[], const</pre>					

1 ticket229.1. MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, ticket-248T. 2 recvtype, root, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 4 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 5 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) 6 INTEGER, INTENT(IN) :: recvcount, root 7 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm 9 TYPE(MPI_Request), INTENT(OUT) :: request 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 12MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 13 RECVTYPE, ROOT, COMM, REQUEST, IERROR) 14<type> SENDBUF(*), RECVBUF(*) 15INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 16 COMM, REQUEST, IERROR ¹⁷ ticket272. [{MPI::Request MPI::Comm::Iscatterv(const void* sendbuf, 18 const int sendcounts[], const int displs[], 19 const MPI::Datatype& sendtype, void* recvbuf, int recvcount, 20const MPI::Datatype& recvtype, int root) const = 0(binding 21deprecated, see Section 15.2 } 22 2324This call starts a nonblocking variant of MPI_SCATTERV (see Section 5.6). 25265.12.5 Nonblocking Gather-to-all 272829 MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, 30 request) 3132 IN sendbuf starting address of send buffer (choice) 33 IN sendcount number of elements in send buffer (non-negative inte-34 ger) 35 IN sendtype data type of send buffer elements (handle) 36 37 OUT recvbuf address of receive buffer (choice) 38 number of elements received from any process (non-IN recvcount 39 negative integer) 40 IN recvtype data type of receive buffer elements (handle) 41 42IN comm communicator (handle) 43

45
46 ticket140.
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
48

communication request (handle)

Unofficial Draft for Comment Only

OUT

request

ticket-248T.

```
1
            \mathbf{2}
                 MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                 comm, request, ierror) BIND(C)
            3
            4
                      TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                      TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
            5
                      INTEGER, INTENT(IN) :: sendcount, recvcount
            6
                      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
            7
                      TYPE(MPI_Comm), INTENT(IN) :: comm
            8
                      TYPE(MPI_Request), INTENT(OUT) :: request
            9
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           10
           11
                 MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
           12
                                COMM, REQUEST, IERROR)
           13
                      <type> SENDBUF(*), RECVBUF(*)
           14
                      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
  ticket272.<sup>15</sup>
                      [{MPI::Request MPI::Comm::Iallgather(const void* sendbuf,
           16
                                 int sendcount, const MPI::Datatype& sendtype, void* recvbuf,
           17
                                 int recvcount, const MPI::Datatype& recvtype) const = O(binding
           18
           19
                                 deprecated, see Section 15.2 }
           20
           21
                     This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).
           22
           23
           24
                 MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm,
           25
                                 request)
           26
                   IN
                             sendbuf
                                                         starting address of send buffer (choice)
           27
                   IN
                              sendcount
                                                          number of elements in send buffer (non-negative inte-
           28
                                                         ger)
           29
           30
                   IN
                             sendtype
                                                         data type of send buffer elements (handle)
           31
                   OUT
                              recvbuf
                                                         address of receive buffer (choice)
           32
                   IN
                              recvcounts
                                                         non-negative integer array (of length group size) con-
           33
                                                          taining the number of elements that are received from
           34
                                                          each process
           35
           36
                   IN
                              displs
                                                         integer array (of length group size). Entry i specifies
           37
                                                          the displacement (relative to recvbuf) at which to place
           38
                                                          the incoming data from process i
           39
                   IN
                              recvtype
                                                         data type of receive buffer elements (handle)
           40
                   IN
                              comm
                                                         communicator (handle)
           41
           42
                   OUT
                             request
                                                          communication request (handle)
           43
           44
  ticket140.
                 int MPI_Iallgatherv(const void* sendbuf, int sendcount,
           45
  ticket140.
                                MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
           46
  ticket140.
                                 const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
           47
                                MPI_Request* request)
ticket229.1. 48
ticket-248T.
```

```
1
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                         2
               recvtype, comm, request, ierror) BIND(C)
                                                                                         3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         4
    INTEGER, INTENT(IN) :: sendcount
                                                                                         5
                                                                                         6
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                         7
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                         9
                                                                                         10
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                         11
MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                         12
               RECVTYPE, COMM, REQUEST, IERROR)
                                                                                         13
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         14
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
                                                                                         15
    REQUEST, IERROR
                                                                                         <sup>16</sup> ticket272.
                                                                                         17
    [{MPI::Request MPI::Comm::Iallgatherv(const void* sendbuf,
                                                                                         18
               int sendcount, const MPI::Datatype& sendtype, void* recvbuf,
                                                                                         19
               const int recvcounts[], const int displs[],
                                                                                         20
               const MPI::Datatype& recvtype) const = 0(binding deprecated, see
                                                                                         21
               Section 15.2 }
                                                                                         22
                                                                                         23
    This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7).
                                                                                         24
                                                                                         25
5.12.6 Nonblocking All-to-All Scatter/Gather
                                                                                         26
                                                                                         27
                                                                                         28
MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)
                                                                                         29
                                                                                         30
                                                                                         31
  IN
            sendbuf
                                       starting address of send buffer (choice)
                                                                                         32
  IN
            sendcount
                                       number of elements sent to each process (non-negative
                                                                                         33
                                       integer)
                                                                                         34
  IN
            sendtype
                                       data type of send buffer elements (handle)
                                                                                         35
                                                                                         36
  OUT
            recvbuf
                                       address of receive buffer (choice)
                                                                                         37
  IN
                                       number of elements received from any process (non-
            recycount
                                                                                         38
                                       negative integer)
                                                                                         39
  IN
                                       data type of receive buffer elements (handle)
                                                                                         40
            recvtype
                                                                                         41
  IN
            comm
                                       communicator (handle)
                                                                                         42
  OUT
                                       communication request (handle)
           request
                                                                                         43
                                                                                         44
int MPI_Ialltoall(const void* sendbuf, int sendcount,
                                                                                         45 ticket140.
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                         46
               MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

```
Unofficial Draft for Comment Only
```

48 ticket-248T.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18	<pre>MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,</pre>				
19 20 21 22 23 24 25	Section 15.2) }] This call starts a nonblocking variant of MPI_ALLTOALL (see Section 5.8). MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)				
26	IN	sendbuf	starting address of send buffer (choice)		
27 ticket0. ²⁸	IN	sendcounts	non-negative integer array (of length group size) speci- fying the number of elements to send to each [processor]rank		
29 30 31 32	IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j		
32	IN	sendtype	data type of send buffer elements (handle)		
34	OUT	recvbuf	address of receive buffer (choice)		
$^{35}_{36}$ ticket $0. \frac{^{37}}{^{38}}$	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each [processor]rank		
39 40 41	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i		
42	IN	recvtype	data type of receive buffer elements (handle)		
43	IN	comm	communicator (handle)		
44 45	OUT	request	communication request (handle)		
46 ticket140. 47 ticket140. 48 ticket140. ticket140.	<pre>int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const</pre>				

```
int recvcounts[], const int rdispls[], MPI_Datatype recvtype,
                                                                                     ^{1} ticket 140.
                                                                                      2
              MPI_Comm comm, MPI_Request *request)
                                                                                     <sup>3</sup> ticket229.1.
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                      _{4} ticket-248T.
              rdispls, recvtype, comm, request, ierror) BIND(C)
                                                                                     5
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                     6
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                      7
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                      8
    recvcounts(*), rdispls(*)
                                                                                      9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                     10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                     12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     13
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
                                                                                     14
                                                                                     15
              RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
                                                                                     16
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     17
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                     18
    RECVTYPE, COMM, REQUEST, IERROR
                                                                                     19 ticket272.
    [{MPI::Request MPI::Comm::Ialltoallv(const void* sendbuf,
                                                                                     20
              const int sendcounts[], const int sdispls[],
                                                                                     21
              const MPI::Datatype& sendtype, void* recvbuf,
                                                                                     22
              const int recvcounts[], const int rdispls[],
                                                                                     23
              const MPI::Datatype& recvtype) const = O(binding deprecated, see
                                                                                     24
              Section 15.2) }
                                                                                     25
                                                                                     26
                                                                                     27
    This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 5.8).
                                                                                     28
                                                                                     29
                                                                                     30
                                                                                     31
                                                                                     32
                                                                                     33
                                                                                     34
                                                                                     35
                                                                                     36
                                                                                     37
                                                                                     38
                                                                                     39
                                                                                     40
                                                                                     41
```

1 2	MPI_IALL	.TOALLW(sendbuf, sen recvtypes, comm,	dcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, request)
3	IN	sendbuf	starting address of send buffer (choice)
- 5 6 7	IN	sendcounts	integer array (of length group size) specifying the num- ber of elements to send to each [processor]rank (array ticket0. of non-negative integers)
8 9 10 11	IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
12 13 14 15	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
16	OUT	recvbuf	address of receive buffer (choice)
17 18 19	IN	recvcounts	integer array (of length group size) specifying the num- ber of elements that can be received from each [processor]rankticket0. (array of non-negative integers)
20 21 22 23 24	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
28	IN	comm	communicator (handle)
29 30 21	OUT	request	communication request (handle)
31 ticket140. 32 ticket140. 33 ticket140. 34 ticket140. 35 ticket140. 36 ticket140. 37 ticket229.1. 38 ticket229.1. 38 ticket-248T. 39 40 41 42 43 44 45 46 47 48	 int sdispls[], const MPI_Datatype sendtypes[], void* rec const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *re MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request, ierror) B TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), recvtypes(*) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 		<pre>const MPI_Datatype sendtypes[], void* recvbuf, counts[], const int rdispls[], const recvtypes[], MPI_Comm comm, MPI_Request *request) dcounts, sdispls, sendtypes, recvbuf, lispls, recvtypes, comm, request, ierror) BIND(C) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf YNCHRONOUS :: recvbuf YNCHRONOUS :: sendcounts(*), sdispls(*), *) ENT(IN), ASYNCHRONOUS :: sendtypes(*), IN) :: comm NT(OUT) :: request</pre>

```
1
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                          \mathbf{2}
               RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
                                                                                          3
    <type> SENDBUF(*), RECVBUF(*)
                                                                                          4
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
                                                                                          5
    RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
                                                                                          <sub>6</sub> ticket272.
    [{MPI::Request MPI::Comm::Ialltoallw(const void* sendbuf, const int
                                                                                          7
               sendcounts[], const int sdispls[], const MPI::Datatype
                                                                                          8
               sendtypes[], void* recvbuf, const int recvcounts[], const int
                                                                                          9
               rdispls[], const MPI::Datatype recvtypes[]) const = 0(binding
                                                                                          10
               deprecated, see Section 15.2 }
                                                                                          11
                                                                                          12
                                                                                          13
    This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 5.8).
                                                                                          14
                                                                                          15
5.12.7 Nonblocking Reduce
                                                                                          16
                                                                                          17
                                                                                          18
MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)
                                                                                          19
            sendbuf
  IN
                                       address of send buffer (choice)
                                                                                          20
  OUT
            recvbuf
                                                                                          21
                                       address of receive buffer (choice, significant only at
                                                                                          22
                                       root)
                                                                                          23
  IN
            count
                                       number of elements in send buffer (non-negative inte-
                                                                                          24
                                       ger)
                                                                                          25
  IN
            datatype
                                       data type of elements of send buffer (handle)
                                                                                          26
  IN
            op
                                       reduce operation (handle)
                                                                                          27
                                                                                          28
  IN
            root
                                       rank of root process (integer)
                                                                                          29
  IN
                                       communicator (handle)
            comm
                                                                                          30
  OUT
                                       communication request (handle)
           request
                                                                                          ^{31}
                                                                                          32
int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count,
                                                                                          ^{33} ticket 140.
                                                                                          34
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
               MPI_Request *request)
                                                                                          35
                                                                                          <sub>36</sub> ticket-248T.
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                          37
               ierror) BIND(C)
                                                                                          38
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                          39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                          40
    INTEGER, INTENT(IN) :: count, root
                                                                                          41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                          42
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                          43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          44
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                          45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          46
                                                                                          47
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
                                                                                          48
               IERROR)
```

	2	224		CHAPTER 5. COLLECTIVE COMMUNICATION	
	1	•	ype> SENDBUF(*), R		
ticket272.	2	INT	TEGER COUNT, DATAT	YPE, OP, ROOT, COMM, REQUEST, IERROR	
	4	[{N	MPI::Request MPI::(<pre>Comm::Ireduce(const void* sendbuf, void* recvbuf,</pre>	
	5			<pre>const MPI::Datatype& datatype, const MPI::Op& op,</pre>	
	6		int root) c	<pre>onst = 0(binding deprecated, see Section 15.2) }</pre>	
	7	1			
	8	Thi	is call starts a nonblo	cking variant of MPI_REDUCE (see Section 5.9.1).	
	9 10	4	dvice to implementors	5. The implementation is explicitly allowed to use different	
	10		* · · · · · · · · · · · · · · · · · · ·	and nonblocking reduction operations that might change the	
	12			the operations. However, as for MPI_REDUCE, it is strongly	
	13			_IREDUCE be implemented so that the same result be obtained	
	14			s applied on the same arguments, appearing in the same order.	
	15	Note that this may prevent optimizations that take advantage of the physical location of processes. (<i>End of advice to implementors.</i>)			
	16 17	of	processes. (End of a	dvice to implementors.)	
	18	A	dvice to users. For o	perations which are not truly associative, the result delivered	
	19	up	oon completion of the	nonblocking reduction may not exactly equal the result deliv-	
	20			luction, even when specifying the same arguments in the same	
	21	or	der. (End of advice to	o users.)	
	22	- 10.0	Nambla altimm All Day	1	
	23 24	5.12.8	Nonblocking All-Rec	luce	
	24				
		MPI_IA	LLREDUCE(sendbuf, r	ecvbuf, count, datatype, op, comm, request)	
	27	IN	sendbuf	starting address of send buffer (choice)	
	28 29	OUT	recvbuf	starting address of receive buffer (choice)	
	30	IN	count	number of elements in send buffer (non-negative inte-	
	31		count	ger)	
	32	IN	datatype	data type of elements of send buffer (handle)	
	33 34	IN	ор	operation (handle)	
	35	IN	comm	communicator (handle)	
	36	OUT	request	communication request (handle)	
	37 38				
ticket 140.	39	int MP]		void* sendbuf, void* recvbuf, int count,	
	40		••	e datatype, MPI_Op op, MPI_Comm comm,	
ticket-248T.	41		MPI_Request	*request)	
		MPI_Ial		recvbuf, count, datatype, op, comm, request,	
	43 44	TUT	ierror) BIN		
	45			.), INTENT(IN), ASYNCHRONOUS :: sendbuf .), ASYNCHRONOUS :: recvbuf	
	46		<pre>rE(*), DIMENSION(. rEGER, INTENT(IN)</pre>		
	47			INTENT(IN) :: datatype	
	48		PE(MPI_Op), INTENT	• •	

TYPI	E(MPI_Comm), INTENT(IN) ::	comm	1	
	E(MPI_Request), INTENT(OUT		2	
	EGER, OPTIONAL, INTENT(OUT		3	
			4	
MPI_IAL		COUNT, DATATYPE, OP, COMM, REQUEST,	5	
	IERROR)			
• •	<pre>De> SENDBUF(*), RECVBUF(*)</pre>		7	
INTI	EGER COUNT, DATATYPE, OP,	COMM, REQUEST, IERROR	8 ticket272.	
[{ MI	PI::Request MPI::Comm::Tal	lreduce(const void* sendbuf,	9	
[[····		ount, const MPI::Datatype& datatype,	10	
	const MPI::Op& op) const = 0(binding deprecated, see Section 15.2) }			
			12	
]			13	
This	call starts a nonblocking varia	nt of MPI_ALLREDUCE (see Section $5.9.6$).	14	
			15	
5.12.9	Nonblocking Reduce-Scatter w	rith Equal Blocks	16	
	-		17	
			18	
MPI IRE	DUCE SCATTER BLOCK(send	buf, recvbuf, recvcount, datatype, op, comm, request)	19	
			20	
	н. с		21	
IN	sendbuf	starting address of send buffer (choice)	22	
OUT	recvbuf	starting address of receive buffer (choice)	23	
IN	recvcount	element count per block (non-negative integer)	24	
			25	
IN	datatype	data type of elements of send and receive buffers (han-	26	
		dle)	27	
IN	ор	operation (handle)	28	
IN	comm	communicator (handle)	29	
OUT	request	communication request (handle)	30	
001	request	communication request (handle)	31	
			32	
int MPI.		st void* sendbuf, void* recvbuf,	$_{33}^{\circ}$ ticket 140.	
		atatype datatype, MPI_Op op,	34	
	MPI_Comm comm, MPI_R	equest *request)	35 ticket-248T.	
MPI Irea	luce scatter block(sendbuf	, recvbuf, recvcount, datatype, op, comm,	36	
_	request, ierror) BIN	· · · ·	37	
TYPI	-	T(IN), ASYNCHRONOUS :: sendbuf	38	
	E(*), DIMENSION(), ASYNC		39	
	EGER, INTENT(IN) :: recvc		40	
	E(MPI_Datatype), INTENT(IN		41	
	E(MPI_Op), INTENT(IN) ::	• •	42	
	E(MPI_Comm), INTENT(IN) ::	-	43	
	E(MPI_Request), INTENT(OUT		44	
	EGER, OPTIONAL, INTENT(OUT	-	45	
			46	
MPI_IREI		, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,	47	
REQUEST, IERROR)				

1 <type> SENDBUF(*), RECVBUF(*) 2 INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR ticket $272._3$ [{MPI::Request MPI::Comm::Ireduce_scatter_block(const void* sendbuf, 4 void* recvbuf, int recvcount, const MPI::Datatype& datatype, 5 const MPI::Op& op) const = O(binding deprecated, see Section 15.2) } 6 7 8 This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-9 tion 5.10.1). 10 11 5.12.10 Nonblocking Reduce-Scatter 1213 14MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request) 15IN sendbuf starting address of send buffer (choice) 1617 OUT recvbuf starting address of receive buffer (choice) 18 IN recvcounts non-negative integer array specifying the number of 19 elements in result distributed to each process. Array 20must be identical on all calling processes. 21IN data type of elements of input buffer (handle) datatype 22 23IN operation (handle) op 24 IN comm communicator (handle) 25OUT communication request (handle) 26request 27ticket140. 28 int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, const ticket140. 29 int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request) 30 ticket229.1. 31 MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, ticket-248T. 32 request, ierror) BIND(C) 33 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 34 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 35 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*) 36 TYPE(MPI_Datatype), INTENT(IN) :: datatype 37 TYPE(MPI_Op), INTENT(IN) :: op 38 TYPE(MPI_Comm), INTENT(IN) :: comm 39 TYPE(MPI_Request), INTENT(OUT) :: request 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 43 REQUEST, IERROR) 44 <type> SENDBUF(*), RECVBUF(*) 45 INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR ticket272. 46 [{MPI:::Request MPI::Comm::Ireduce_scatter(const void* sendbuf, 47 void* recvbuf, int recvcounts[], 48

1 const MPI::Datatype& datatype, const MPI::Op& op) $\mathbf{2}$ const = 0 (binding deprecated, see Section 15.2) } 3 4 This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2). 5 6 5.12.11 Nonblocking Inclusive Scan 7 8 9 MPI_ISCAN(sendbuf, recvbuf, count, datatype, op, comm, request) 10 11 IN sendbuf starting address of send buffer (choice) 12OUT recvbuf starting address of receive buffer (choice) 13 IN count number of elements in input buffer (non-negative in-1415teger) 16IN datatype data type of elements of input buffer (handle) 17IN operation (handle) op 18 IN comm communicator (handle) 19 20OUT request communication request (handle) 2122 int MPI_Iscan(const void* sendbuf, void* recvbuf, int count, ticket140. 23 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 24 MPI_Request *request) 25ticket-248T. 26MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) BIND(C) 27TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 28 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 29 30 INTEGER, INTENT(IN) :: count 31TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op 32 33 TYPE(MPI_Comm), INTENT(IN) :: comm 34 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 37 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR ³⁹ ticket272. [{MPI::Request MPI::Intracomm::Iscan(const void* sendbuf, 40 41 void* recvbuf, int count, const MPI::Datatype& datatype, 42const MPI::Op& op) const(binding deprecated, see Section 15.2) } 43 44This call starts a nonblocking variant of MPI_SCAN (see Section 5.11). 454647

```
228
                                                     CHAPTER 5. COLLECTIVE COMMUNICATION
            1
                          Nonblocking Exclusive Scan
                 5.12.12
            \mathbf{2}
            3
            4
                 MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
            5
                   IN
                            sendbuf
                                                         starting address of send buffer (choice)
            6
                   OUT
            7
                             recvbuf
                                                         starting address of receive buffer (choice)
            8
                   IN
                                                         number of elements in input buffer (non-negative in-
                             count
            9
                                                         teger)
            10
                   IN
                            datatype
                                                         data type of elements of input buffer (handle)
           11
                   IN
                                                         operation (handle)
           12
                             op
           13
                   IN
                                                         intracommunicator (handle)
                             comm
            14
                   OUT
                                                         communication request (handle)
                            request
            15
           16
  ticket140. 17
                 int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,
                                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
            18
                                MPI_Request *request)
            19
ticket-248T.
            20
                 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
           21
                                BIND(C)
           22
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
           23
                     TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
           24
                     INTEGER, INTENT(IN) :: count
           25
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           26
                     TYPE(MPI_Op), INTENT(IN) :: op
           27
                     TYPE(MPI_Comm), INTENT(IN) :: comm
           28
                     TYPE(MPI_Request), INTENT(OUT) :: request
           29
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           30
                 MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
           ^{31}
                      <type> SENDBUF(*), RECVBUF(*)
           32
                      INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
           33
  ticket272.
           34
                     [{MPI::Request MPI::Intracomm::Iexscan(const void* sendbuf, void*
           35
                                recvbuf, int count, const MPI::Datatype& datatype, const
           36
                                MPI::Op& op) const(binding deprecated, see Section 15.2) }
           37
           38
                     This call starts a nonblocking variant of MPI_EXSCAN (see Section 5.11.2).
           39
            40
           41
                 5.13
                         Correctness
           42
           43
                 A correct, portable program must invoke collective communications so that deadlock will not
           44
                 occur, whether collective communications are synchronizing or not. The following examples
           45
                 illustrate dangerous use of collective routines on intracommunicators.
           46
           47
                 Example 5.25
           48
                     The following is erroneous.
```

<pre>switch(rank) {</pre>	1
case 0:	2
<pre>MPI_Bcast(buf1, count, type, 0, comm);</pre>	3
<pre>MPI_Bcast(buf2, count, type, 1, comm);</pre>	4
break;	5
case 1:	6
<pre>MPI_Bcast(buf2, count, type, 1, comm);</pre>	7
<pre>MPI_Bcast(buf1, count, type, 0, comm);</pre>	8
break;	9
}	10
	11

We assume that the group of comm is $\{0,1\}$. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

Example 5.26

The following is erroneous.

```
switch(rank) {
   case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
   case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
   case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 2:
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
   case 2:
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 3:
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 3:
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
   case 3:
   case
```

}

Assume that the group of comm0 is $\{0,1\}$, of comm1 is $\{1, 2\}$ and of comm2 is $\{2,0\}$. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependencies occur. Nonblocking collective operations can alleviate this issue.

Example 5.27

The following is erroneous.

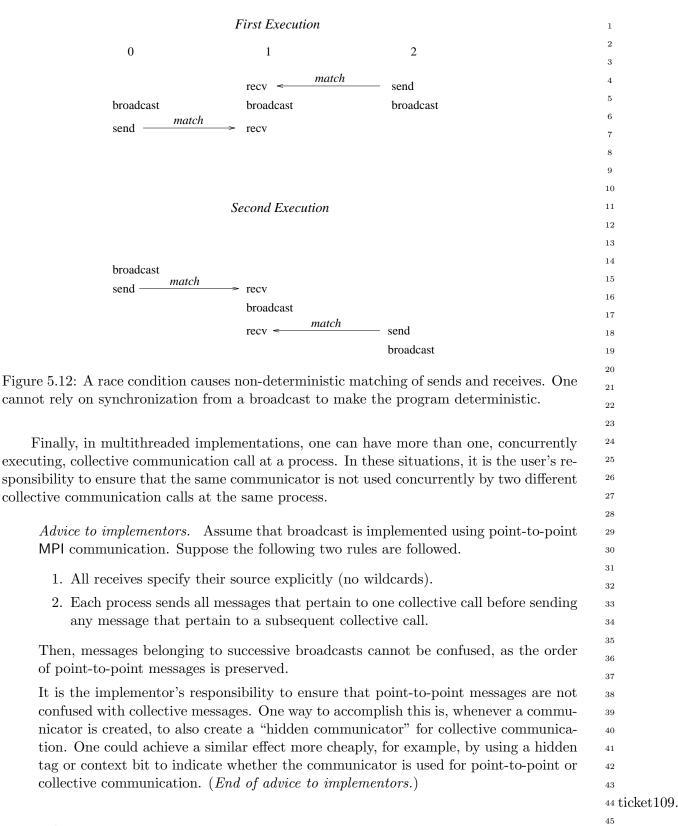
 24

 31

 $41 \\ 42$

```
1
      switch(rank) {
\mathbf{2}
          case 0:
3
               MPI_Bcast(buf1, count, type, 0, comm);
4
               MPI_Send(buf2, count, type, 1, tag, comm);
5
               break:
6
          case 1:
7
               MPI_Recv(buf2, count, type, 0, tag, comm, status);
8
               MPI_Bcast(buf1, count, type, 0, comm);
9
               break;
10
     }
11
          Process zero executes a broadcast, followed by a blocking send operation. Process one
12
      first executes a blocking receive that matches the send, followed by broadcast call that
13
      matches the broadcast of process zero. This program may deadlock. The broadcast call on
14
      process zero may block until process one executes the matching broadcast call, so that the
15
      send is not executed. Process one will definitely block on the receive and so, in this case,
16
     never executes the broadcast.
17
          The relative order of execution of collective operations and point-to-point operations
18
     should be such, so that even if the collective operations and the point-to-point operations
19
      are synchronizing, no deadlock will occur.
20
21
      Example 5.28
22
          An unsafe, non-deterministic program.
23
^{24}
      switch(rank) {
25
          case 0:
26
               MPI_Bcast(buf1, count, type, 0, comm);
27
               MPI_Send(buf2, count, type, 1, tag, comm);
28
               break;
29
          case 1:
30
               MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
31
               MPI_Bcast(buf1, count, type, 0, comm);
32
               MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
33
               break;
34
          case 2:
35
               MPI_Send(buf2, count, type, 1, tag, comm);
36
               MPI_Bcast(buf1, count, type, 0, comm);
37
               break;
38
      }
39
40
          All three processes participate in a broadcast. Process 0 sends a message to process
41
      1 after the broadcast, and process 2 sends a message to process 1 before the broadcast.
42
      Process 1 receives before and after the broadcast, with a wildcard source argument.
43
          Two possible executions of this program, with different matchings of sends and receives,
44
      are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that
45
      a send executed after the broadcast is received at another node before the broadcast. This
```

example illustrates the fact that one should not rely on collective communication functions
 to have particular synchronization effects. A program that works correctly only when the
 first execution occurs (only when broadcast is synchronizing) is erroneous.



Example 5.29

Blocking and nonblocking collective operations can be interleaved, i.e., a blocking collective operation can be posted even if there is a nonblocking collective operation outstanding.

Unofficial Draft for Comment Only

46

47

```
1
     MPI_Request req;
\mathbf{2}
3
     MPI_Ibarrier(comm, &req);
4
     MPI_Bcast(buf1, count, type, 0, comm);
\mathbf{5}
     MPI_Wait(&req, MPI_STATUS_IGNORE);
6
          Each process starts a nonblocking barrier operation, participates in a blocking broad-
7
     cast and then waits until every other process started the barrier operation. This ef-
8
     fectively turns the broadcast into a synchronizing broadcast with possible communica-
9
     tion/communication overlap (MPI_Bcast is allowed, but not required to synchronize).
10
11
     Example 5.30
12
          The starting order of collective operations on a particular communicator defines their
13
     matching. The following example shows an erroneous matching of different collective oper-
14
     ations on the same communicator.
15
16
     MPI_Request req;
17
     switch(rank) {
18
          case 0:
19
              /* erroneous matching */
20
              MPI_Ibarrier(comm, &req);
21
              MPI_Bcast(buf1, count, type, 0, comm);
22
              MPI_Wait(&req, MPI_STATUS_IGNORE);
23
              break;
24
          case 1:
25
              /* erroneous matching */
26
              MPI_Bcast(buf1, count, type, 0, comm);
27
              MPI_Ibarrier(comm, &req);
28
              MPI_Wait(&reg, MPI_STATUS_IGNORE);
29
              break;
30
     }
^{31}
32
         This ordering would match MPI_lbarrier on rank 0 with MPI_Bcast on rank 1 which is
33
     erroneous and the program behavior is undefined. However, if such an order is required, the
34
     user must create different duplicate communicators and perform the operations on them.
35
     If started with two processes, the following program would be correct:
36
37
     MPI_Request req;
38
     MPI_Comm dupcomm;
39
     MPI_Comm_dup(comm, &dupcomm);
40
     switch(rank) {
41
          case 0:
42
              MPI_Ibarrier(comm, &req);
              MPI_Bcast(buf1, count, type, 0, dupcomm);
43
44
              MPI_Wait(&req, MPI_STATUS_IGNORE);
45
              break;
46
          case 1:
47
              MPI_Bcast(buf1, count, type, 0, dupcomm);
48
              MPI_Ibarrier(comm, &req);
```

```
MPI_Wait(&req, MPI_STATUS_IGNORE);
break;
```

}

Advice to users. The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

Example 5.31

Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;
```

```
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        MPI_Send(buf, count, dtype, 1, tag, comm);
        break;
    case 1:
        MPI_Ibarrier(comm, &req);
        MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
```

```
}
```

The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls eventually return.

Example 5.32

Blocking and nonblocking collective operations do not match. The following example is erroneous.

```
MPI_Request req;
```

```
switch(rank) {
                                                                                    38
                                                                                    39
    case 0:
      /* erroneous false matching of Alltoall and Ialltoall */
                                                                                    40
                                                                                    41
      MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
                                                                                    42
      MPI_Wait(&req, MPI_STATUS_IGNORE);
      break;
                                                                                    43
                                                                                    44
    case 1:
      /* erroneous false matching of Alltoall and Ialltoall */
                                                                                    45
                                                                                    46
      MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
                                                                                    47
      break;
                                                                                    48
}
```

 $\mathbf{2}$

```
1
     Example 5.33
\mathbf{2}
          Collective and point-to-point requests can be mixed in functions that enable multiple
3
     completions. If started with two processes, the following program is valid.
4
     MPI_Request reqs[2];
5
6
     switch(rank) {
7
8
          case 0:
9
            MPI_Ibarrier(comm, &reqs[0]);
            MPI_Send(buf, count, dtype, 1, tag, comm);
10
            MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
11
            break;
12
          case 1:
13
            MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
14
            MPI_Ibarrier(comm, &reqs[1]);
15
            MPI_Waitall(2, regs, MPI_STATUSES_IGNORE);
16
            break;
17
     }
18
19
          The Waitall call returns only after the barrier and the receive completed.
20
21
     Example 5.34
22
          Multiple nonblocking collective operations can be outstanding on a single communicator
23
     and match in order.
24
25
     MPI_Request reqs[3];
26
27
     compute(buf1);
28
     MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
29
     compute(buf2);
30
     MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
^{31}
     compute(buf3);
32
     MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
33
     MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
34
35
           Advice to users. Pipelining and double-buffering techniques can efficiently be used
36
           to overlap computation and communication. However, having too many outstanding
37
           requests might have a negative impact on performance. (End of advice to users.)
38
39
                                      The use of pipelining may generate many outstanding
           Advice to implementors.
40
           requests. A high-quality hardware-supported implementation with limited resources
41
           should be able to fall back to a software implementation if its resources are exhausted.
42
           In this way, the implementation could limit the number of outstanding requests only
43
           by the available memory. (End of advice to implementors.)
44
45
46
     Example 5.35
47
48
```

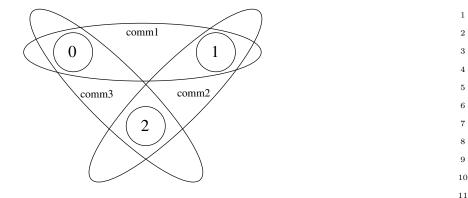


Figure 5.13: Example with overlapping communicators.

Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 5.13). The following example is started with three processes and three communicators. The first communicator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2 and comm3 spans ranks 0 and 2. It is not possible to perform a blocking collective operation on all communicators because there exists no deadlock-free order to invoke them. However, nonblocking collective operations can easily be used to achieve this task.

```
MPI_Request reqs[2];
```

```
switch(rank) {
    case 0:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
                                                                                 27
      break;
    case 1:
                                                                                 29
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
      break;
    case 2:
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
                                                                                 34
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
      break;
                                                                                 36
}
                                                                                 37
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```

Advice to users. This method can be useful if overlapping neighboring regions (halo or ghost zones) are used in collective operations. The sequence of the two calls in each process is irrelevant because the two nonblocking operations are performed on different communicators. (End of advice to users.)

Example 5.36

The progress of multiple outstanding nonblocking collective operations is completely independent.

Unofficial Draft for Comment Only

1213 14

15

16

17

18

19

2021

22 23 24

25

26

28

30

31

32 33

35

38 39

40

41

42

43 4445

46

47

```
1
     MPI_Request reqs[2];
\mathbf{2}
3
     compute(buf1);
4
     MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
\mathbf{5}
     compute(buf2);
6
     MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
7
     MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
8
     /* nothing is known about the status of the first bcast here */
9
     MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
10
11
          Finishing the second MPI_IBCAST is completely independent of the first one. This
     means that it is not guaranteed that the first broadcast operation is finished or even started
12
     after the second one is completed via reqs[1].
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Chapter 6

Groups, Contexts, Communicators, and Caching

6.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [55] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

 24

6.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- Groups of processes,
- Virtual topologies,
- Attribute caching,
- Communicators.

¹³ ¹⁴ **Communicators** (see [20, 53, 58]) encapsulate all of these ideas in order to provide the ¹⁵ appropriate scope for all communication operations in MPI. Communicators are divided ¹⁶ into two kinds: intra-communicators for operations within a single group of processes and ¹⁷ inter-communicators for operations between two groups of processes.

¹⁹ Caching. Communicators (see below) provide a "caching" mechanism that allows one to ²⁰ associate new attributes with communicators, on a par with MPI built-in features. This ²¹ can be used by advanced users to adorn communicators further, and by MPI to implement ²² some communicator functions. For example, the virtual-topology functions described in ²³ Chapter 7 are likely to be supported this way.

24

1

2

3

4 5

6

7 8

9 10

11

12

18

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

31

36

37

38

39

40

41

42

43

44

45 46

Intra-communicators. The most commonly used means for message passing in MPI is via
 intra-communicators. Intra-communicators contain an instance of a group, contexts of
 communication for both point-to-point and collective communication, and the ability to
 include virtual topology and other attributes. These features work as follows:

• **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.

- **Groups** define the participants in the communication (see above) of a communicator.
- 47 48

- A virtual topology defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 7 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. This practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. Users who are satisfied with this practice can plug in MPI_COMM_WORLD wherever a communicator argument is required, and can consequently disregard the rest of this chapter. (End of advice to users.)

Inter-communicators. The discussion has dealt so far with intra-communication: communication within a group. MPI also supports inter-communication: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- **Contexts** provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in an related manner to intracommunicators. Users who do not need inter-communication in their applications can safely

Unofficial Draft for Comment Only

 24

 31

ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

6.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

6.2.1 Groups

1

 $\mathbf{2}$

3 4

5 6

> 7 8

9

20 21

22

23 24

25

26

27

28

34

35

¹⁰ A **group** is an ordered set of process identifiers (henceforth processes); processes are ¹¹ implementation-dependent objects. Each process in a group is associated with an inte-¹² ger **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque ¹³ **group objects**, and hence cannot be directly transferred from one process to another. A ¹⁴ group is used within a communicator to describe the participants in a communication "uni-¹⁵ verse" and to rank such participants (thus giving them unique names within that "universe" ¹⁶ of communication).

There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no
 members. The predefined constant MPI_GROUP_NULL is the value used for invalid group
 handles.

Advice to users. MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (*End of advice to users.*)

Advice to implementors. A group may be represented by a virtual-to-real processaddress-translation table. Each communicator object (see below) would have a pointer to such a table.

Simple implementations of MPI will enumerate groups, such as in a table. However,
 more advanced data structures make sense in order to improve scalability and memory
 usage with large numbers of processes. Such implementations are possible with MPI.
 (End of advice to implementors.)

6.2.2 Contexts

A context is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-topoint operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

 $41 \\ 42$

43

44

45

46

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicatorgenerating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

6.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 7), communicators may also "cache" additional information (see Section 6.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

6.2.4 Predefined Intra-Communicators

An initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the 37 computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a 3839 communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynami-40 41 cally join an MPI execution, it may be the case that a process starts an MPI computation 42without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately 4344communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups 45in different processes.

All MPI implementations are required to provide the MPI_COMM_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to this communicator does not appear as a pre-defined constant, but it may be accessed using

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28 29

30

31

32 33

34

35

```
1
                 MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the
            \mathbf{2}
                 process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither
            3
                 does MPI specify the function of the host process, if any. Other implementation-dependent,
            4
                 predefined communicators may also be provided.
            5
            6
                 6.3
                       Group Management
            7
            8
                 This section describes the manipulation of process groups in MPI. These operations are
            9
                 local and their execution does not require interprocess communication.
            10
            11
                 6.3.1 Group Accessors
           12
            13
           14
                 MPI_GROUP_SIZE(group, size)
           15
            16
                                                         group (handle)
                   IN
                             group
            17
                   OUT
                             size
                                                         number of processes in the group (integer)
            18
            19
                 int MPI_Group_size(MPI_Group group, int *size)
           20
ticket-248T.
            21
                 MPI_Group_size(group, size, ierror) BIND(C)
           22
                      TYPE(MPI_Group), INTENT(IN) :: group
           23
                      INTEGER, INTENT(OUT) :: size
           ^{24}
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           25
                 MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
            26
                      INTEGER GROUP, SIZE, IERROR
           27
           28
                 {int MPI:::Group:::Get_size() const(binding deprecated, see Section 15.2) }
           29
           30
           ^{31}
                 MPI_GROUP_RANK(group, rank)
           32
                   IN
                             group
                                                         group (handle)
           33
           34
                   OUT
                             rank
                                                         rank of the calling process in group,
                                                                                                     or
           35
                                                         MPI_UNDEFINED if the process is not a member (in-
           36
                                                         teger)
           37
           38
                 int MPI_Group_rank(MPI_Group group, int *rank)
ticket-248T. 39
                 MPI_Group_rank(group, rank, ierror) BIND(C)
            40
                      TYPE(MPI_Group), INTENT(IN) ::
                                                          group
           41
                      INTEGER, INTENT(OUT) :: rank
           42
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           43
           44
                 MPI_GROUP_RANK(GROUP, RANK, IERROR)
           45
                      INTEGER GROUP, RANK, IERROR
           46
           47
                 {int MPI::Group::Get_rank() const(binding deprecated, see Section 15.2) }
           48
```

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2) ¹				
IN	group1	group1 (handle)	2	
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)	3 4	
IN	ranks1	array of zero or more valid ranks in group1	5	
IN	group2	group2 (handle)	6	
OUT	ranks2	array of corresponding ranks in group2,	7	
001	1011652	MPI_UNDEFINED when no correspondence exists.	8 9	
			10	
int MPI_(Group_translate_ranks (MP)	I_Group group1, int n, const int *ranks1,	11 ticket 140.	
	MPI_Group group2, in	t *ranks2)	12 	
MPI Grout	o translate ranks(group1.	n, ranks1, group2, ranks2, ierror)	$_{13}$ ticket-248T.	
_ 1 1 1	BIND(C)		14 15	
	(MPI_Group), INTENT(IN) :		16	
	GER, INTENT(IN) :: n, rai		17	
	GER, INTENT(OUT) :: rank		18	
INTEC	GER, OPTIONAL, INTENT(OUT)) :: lerror	19	
		N, RANKS1, GROUP2, RANKS2, IERROR)	20	
INTEC	GER GROUP1, N, RANKS1(*),	GROUP2, RANKS2(*), IERROR	21 22	
$\{ \texttt{static } v \}$		e_ranks (const MPI::Group& group1, int n, const MPI::Group& group2,	23	
	24			
<pre>int ranks2[])(binding deprecated, see Section 15.2) }</pre>			25	
This f	26			
in two diffe	27			
of MPI_CO	28 29			
MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.			30	
10001110 111			31	
			32	
	UP_COMPARE(group1, group2		33	
IN	group1	first group (handle)	34 35	
IN	group2	second group (handle)	36	
OUT	result	result (integer)	37	
			38	
int MPI_C	Group_compare(MPI_Group_g	roup1,MPI_Group group2, int *result)	$^{39}_{40}$ ticket-248T.	
MPI_Group	<pre>compare(group1, group2,</pre>	result, ierror) BIND(C)	40 010100 2 10 1 . 41	
	(MPI_Group), INTENT(IN) :		42	
	INTEGER, INTENT(OUT) :: result			
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44			
	P_COMPARE(GROUP1, GROUP2,	-	45	
INTEC	GER GROUP1, GROUP2, RESUL	Γ, IERROR	46 47	
			48	

```
1
                 {static int MPI::Group::Compare(const MPI::Group& group1,
            \mathbf{2}
                                 const MPI::Group& group2) (binding deprecated, see Section 15.2) }
            3
                 MPI_IDENT results if the group members and group order is exactly the same in both groups.
            4
                 This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if
            5
                 the group members are the same but the order is different. MPI_UNEQUAL results otherwise.
            6
            7
                 6.3.2
                        Group Constructors
            8
            9
                 Group constructors are used to subset and superset existing groups. These constructors
            10
                 construct new groups from existing groups. These are local operations, and distinct groups
            11
                 may be defined on different processes; a process may also define a group that does not
            12
                 include itself. Consistent definitions are required when groups are used as arguments in
            13
                 communicator-building functions. MPI does not provide a mechanism to build a group
            14
                 from scratch, but only from other, previously defined groups. The base group, upon which
            15
                 all other groups are defined, is the group associated with the initial communicator
            16
                 MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).
            17
            18
                       Rationale.
                                    In what follows, there is no group duplication function analogous to
            19
                       MPI_COMM_DUP, defined later in this chapter. There is no need for a group dupli-
            20
                       cator. A group, once created, can have several references to it by making copies of
            21
                       the handle. The following constructors address the need for subsets and supersets of
            22
                       existing groups. (End of rationale.)
            23
            24
                                                  Each group constructor behaves as if it returned a new
                       Advice to implementors.
                       group object. When this new group is a copy of an existing group, then one can
            25
            26
                       avoid creating such new objects, using a reference-count mechanism. (End of advice
            27
                       to implementors.)
            28
            29
            30
                 MPI_COMM_GROUP(comm, group)
            ^{31}
                   IN
                                                          communicator (handle)
            32
                             comm
            33
                   OUT
                                                          group corresponding to comm (handle)
                             group
            34
            35
                 int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
            36
ticket-248T.
            37
                 MPI_Comm_group(comm, group, ierror) BIND(C)
            38
                      TYPE(MPI_Comm), INTENT(IN) :: comm
            39
                      TYPE(MPI_Group), INTENT(OUT) :: group
            40
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            41
                 MPI_COMM_GROUP(COMM, GROUP, IERROR)
            42
                      INTEGER COMM, GROUP, IERROR
            43
            44
                 {MPI::Group MPI::Comm::Get_group() const(binding deprecated, see Section 15.2) }
            45
                      MPI_COMM_GROUP returns in group a handle to the group of comm.
            46
            47
            48
```

MPI_GRO	OUP_UNION(group1,	group2, newgroup)	1
IN	group1	first group (handle)	2
IN	group2	second group (handle)	3
OUT	newgroup	union group (handle)	5
001	newgroup	union group (nandic)	6
int MPT	Group union(MPT G	roup group1, MPI_Group group2,	7
	MPI_Group *r		8
MDT Creat	-	•••	⁹ ticket-248T.
		roup2, newgroup, ierror) BIND(C) NT(IN) :: group1, group2	10
	-	NT(OUT) :: newgroup	11
	-	TENT(OUT) :: ierror	12
			14
		ROUP2, NEWGROUP, IERROR) 2, NEWGROUP, IERROR	15
			16
$\{\texttt{static}$	-	roup::Union(const MPI::Group& group1,	17
	const MPI::(Group& group2) (binding deprecated, see Section 15.2) }	18
			19
			20
	JUP_INTERSECTION	l(group1, group2, newgroup)	21 22
IN	group1	first group (handle)	23
IN	group2	second group (handle)	24
OUT	newgroup	intersection group (handle)	25
			26
int MPI_	Group_intersectio	n(MPI_Group group1, MPI_Group group2,	27
	MPI_Group *r	newgroup)	28
MPT Grou	n intersection(gr	oup1, group2, newgroup, ierror) BIND(C)	29 ticket-248T.
	• •	NT(IN) :: group1, group2	30 31
		NT(OUT) :: newgroup	31
	-	TENT(OUT) :: ierror	33
	ID INTERSECTION (CR	OUP1, GROUP2, NEWGROUP, IERROR)	34
_		2, NEWGROUP, IERROR	35
			36
{static	-	roup::Intersect(const MPI::Group& group1,	37
	const MP1::0	Group& group2) (binding deprecated, see Section 15.2) }	38
			39
		roup1, group2, newgroup)	40 41
	(2		42
IN	group1	first group (handle)	43
IN	group2	second group (handle)	44
OUT	newgroup	difference group (handle)	45
			46
int MPI_	Group_difference(MPI_Group group1, MPI_Group group2,	47
MPI_Group *newgroup)			

ticket-248T. 2 3 4 5	ТҮР ТҮР	PE(MPI_Group), INTH	up1, group2, newgroup, ierror) BIND(C) ENT(IN) :: group1, group2 ENT(OUT) :: newgroup ITENT(OUT) :: ierror		
6 7 8		<pre>MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR {static MPI::Group MPI::Group::Difference(const MPI::Group& group1,</pre>			
9 10	{static				
11	The set-	like operations are de	fined as follows:		
13 14		All elements of the fir roup2) not in first.	est group (group1), followed by all elements of second group $% \mathcal{G}(\mathcal{G})$		
15 16 17		ct all elements of the st group.	first group that are also in the second group, ordered as in		
18 19		ace all elements of the first group.	e first group that are not in the second group, ordered as in		
20 21 22 23 24 25	primaril second g	Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative. The new group can be empty, that is, equal to MPI_GROUP_EMPTY.			
26	MPI_GR	OUP_INCL(group, n,	ranks, newgroup)		
27	IN	group	group (handle)		
28 29 30	IN	n	number of elements in array ranks (and size of newgroup) (integer)		
31 32	IN	ranks	ranks of processes in group to appear in newgroup (array of integers)		
33 34 35	OUT	newgroup	new group derived from above, in the order defined by $ranks$ (handle)		
ticket 140. $\frac{^{36}}{_{37}}$	int MPI	Group_incl(MPI_G MPI_Group *	coup group, int n, const int *ranks, newgroup)		
ticket229.1. ³⁸ ticket-248T. ³⁹ 40 41 42 43	TYP INT TYP	<pre>MPI_Group_incl(group, n, ranks, newgroup, ierror) BIND(C) TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: n, ranks(n) TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
44 45 46			RANKS, NEWGROUP, IERROR) IKS(*), NEWGROUP, IERROR		
47 48		<pre>{MPI::Group MPI::Group::Incl(int n, const int ranks[]) const(binding</pre>			

The function MPI_GROUP_INCL creates a group newgroup that consists of the 1 n processes in group with ranks ranks[0],..., ranks[n-1]; the process with rank i in newgroup 2 ticket254. 3 ticket254. is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid 4 rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, 5then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI_GROUP_COMPARE. 6 7 8

MPI_GROUP_EXCL(group, n, ranks, newgroup) 9 10 IN group group (handle) 11 IN number of elements in array ranks (integer) n 12IN ranks array of integer ranks in group not to appear in 13 newgroup 1415OUT new group derived from above, preserving the order newgroup 16 defined by group (handle) 17 18int MPI_Group_excl(MPI_Group group, int n, const int *ranks, ticket140. 19 MPI_Group *newgroup) ²⁰ ticket-248T. MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C) 21TYPE(MPI_Group), INTENT(IN) :: group 22 INTEGER, INTENT(IN) :: n, ranks(n) 23TYPE(MPI_Group), INTENT(OUT) :: newgroup 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) 27INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR 28 {MPI::Group MPI::Group::Excl(int n, const int ranks[]) const/binding 29 deprecated, see Section 15.2 } 30 31

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0],... ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks 34 must be a valid rank in group and all elements must be distinct; otherwise, the program is 35 erroneous. If n = 0, then newgroup is identical to group. 36

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

IN	group	group (handle)	39 40
IN	n	number of triplets in array ranges (integer)	41
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup	42 43 44
OUT	newgroup	new group derived from above, in the order defined by $ranges\ (\mathrm{handle})$	45 46 47

32

33

37

38 30

```
1
                   int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
             2
                                   MPI_Group *newgroup)
ticket-248T. 3
                   MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)
             4
                       TYPE(MPI_Group), INTENT(IN) :: group
             5
                       INTEGER, INTENT(IN) :: n, ranges(3,n)
             6
                       TYPE(MPI_Group), INTENT(OUT) :: newgroup
             7
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
             8
             9
                  MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
             10
                       INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
             11
                   {MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3])
            12
                                   const(binding deprecated, see Section 15.2)
             13
             14
                  If ranges consist of the triplets
             15
                        (first_1, last_1, stride_1), \dots, (first_n, last_n, stride_n)
             16
             17
                   then newgroup consists of the sequence of processes in group with ranks
             18
                        first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots
             19
            20
            21
                        first_n, first_n + stride_n, ..., first_n + \left| \frac{last_n - first_n}{stride_n} \right| stride_n.
            22
            23
            ^{24}
                       Each computed rank must be a valid rank in group and all computed ranks must be
            25
                   distinct, or else the program is erroneous. Note that we may have first_i > last_i, and stride_i
            26
                   may be negative, but cannot be zero.
            27
                       The functionality of this routine is specified to be equivalent to expanding the array
            28
                   of ranges to an array of the included ranks and passing the resulting array of ranks and
            29
                   other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call
            30
                   to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in
            ^{31}
                  the argument ranges.
            32
            33
            34
                   MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)
            35
                    IN
                                                             group (handle)
                               group
            36
                    IN
                                                             number of elements in array ranges (integer)
                               n
            37
            38
                    IN
                                                             a one-dimensional array of integer triplets of the form
                               ranges
            39
                                                             (first rank, last rank, stride), indicating the ranks in
             40
                                                             group of processes to be excluded from the output
            41
                                                             group newgroup.
            42
                     OUT
                                                             new group derived from above, preserving the order
                               newgroup
            43
                                                             in group (handle)
            44
            45
                   int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             46
                                   MPI_Group *newgroup)
ticket-248T. ^{47}
             48
                   MPI_Group_range_excl(group, n, ranges, newgroup, ierror) BIND(C)
```

TYPE(MPI_Group), INTENT(IN) :: group	1
INTEGER, INTENT(IN) :: n, ranges(3,n)	2 3
TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	5
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)	6
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR	7
<pre>{MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3])</pre>	8
<pre>const(binding deprecated, see Section 15.2) }</pre>	9
Each computed rank must be a valid rank in group and all computed ranks must be distinct,	10 11
or else the program is erroneous.	12
The functionality of this routine is specified to be equivalent to expanding the array of	13
ranges to an array of the excluded ranks and passing the resulting array of ranks and other	14
arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to	15
MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in	16
the argument ranges.	17
Advice to users. The range operations do not explicitly enumerate ranks, and	18
therefore are more scalable if implemented efficiently. Hence, we recommend MPI	19
programmers to use them whenenever possible, as high-quality implementations will	20
take advantage of this fact. (End of advice to users.)	21 22
	23
Advice to implementors. The range operations should be implemented, if possible,	24
without enumerating the group members, in order to obtain better scalability (time and space). (<i>End of advice to implementors.</i>)	25
and space). (End of dubice to implementors.)	26
6.3.3 Group Destructors	27
	28
	29
MPI_GROUP_FREE(group)	30
	31 32
INOUT group group (handle)	33
int MPI_Group_free(MPI_Group *group)	34
IIC MF1_Group_free(MF1_Group *group)	35 ticket-248T
MPI_Group_free(group, ierror) BIND(C)	36
TYPE(MPI_Group), INTENT(INOUT) :: group	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
MPI_GROUP_FREE(GROUP, IERROR)	39
INTEGER GROUP, IERROR	40
<pre>{void MPI::Group::Free()(binding deprecated, see Section 15.2) }</pre>	41 42
This operation marks a group object for deallocation. The handle group is set to	43
MPI_GROUP_NULL by the call. Any on-going operation using this group will complete	44
normally.	45
	46
Advice to implementors. One can keep a reference count that is incremented for	47
each call to MPI_COMM_GROUP, MPI_COMM_CREATE and MPI_COMM_DUP, and	48

1 decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group 2 object is ultimately deallocated when the reference count drops to zero. (End of 3 advice to implementors.) 4 56.4 Communicator Management 6 7 This section describes the manipulation of communicators in MPI. Operations that access 8 communicators are local and their execution does not require interprocess communication. 9 Operations that create communicators are collective and may require interprocess commu-10 nication. 11 12Advice to implementors. High-quality implementations should amortize the over-13 heads associated with the creation of communicators (for the same group, or subsets 14thereof) over several calls, by allocating multiple contexts with one collective commu-15nication. (End of advice to implementors.) 1617 6.4.1 Communicator Accessors 18 19The following are all local operations. 2021MPI_COMM_SIZE(comm, size) 22 23IN comm communicator (handle) 24OUT size number of processes in the group of comm (integer) 2526int MPI_Comm_size(MPI_Comm comm, int *size) ticket-248T. 2728 MPI_Comm_size(comm, size, ierror) BIND(C) 29TYPE(MPI_Comm), INTENT(IN) :: comm 30 INTEGER, INTENT(OUT) :: size 31 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 MPI_COMM_SIZE(COMM, SIZE, IERROR) 33 INTEGER COMM, SIZE, IERROR 3435 {int MPI::Comm::Get_size() const(binding deprecated, see Section 15.2) } 36 37 This function is equivalent to accessing the communicator's group with Rationale. 38 MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and 39 then freeing the temporary group via MPI_GROUP_FREE. However, this function is 40 so commonly used, that this shortcut was introduced. (End of rationale.) 41 42This function indicates the number of processes involved in a Advice to users. 43 communicator. For MPI_COMM_WORLD, it indicates the total number of processes 44 available (for this version of MPI, there is no standard way to change the number of 45processes once initialization has taken place). 46This call is often used with the next call to determine the amount of concurrency 47 available for a specific library or program. The following call, MPI_COMM_RANK 48

indicates the rank of the process that calls it in the range from 0...size-1, where size is the return value of MPI_COMM_SIZE.(*End of advice to users.*)

			÷
			4
	MM_RANK(comm, rank)		5
			6
IN	comm	communicator (handle)	7
OUT	rank	rank of the calling process in group of comm (integer)	8
			9
int MPI	_Comm_rank(MPI_Comm comm,	, int *rank)	10
NDT G			$^{11}_{12}$ ticket-248T.
	m_rank(comm, rank, ierrom PE(MPI_Comm), INTENT(IN) :		13
	EGER, INTENT(OUT) :: rar		14
	EGER, OPTIONAL, INTENT(OU		15
1111	LULIT, DI HUNAL, INTENI (DO	51) 161101	16
	IM_RANK(COMM, RANK, IERROF	R)	17
INT	EGER COMM, RANK, IERROR		18
{int MP	'I::Comm::Get rank() const	t(binding deprecated, see Section 15.2) }	19
(· · · · · · · · · · · · · · · · · · ·	20
D	tionale This function is as	uivalent to accessing the communicator's group with	21
	-	re), computing the rank using MPI_GROUP_RANK,	22
		group via MPI_GROUP_FREE. However, this function	23
		shortcut was introduced. (<i>End of rationale.</i>)	24
15	so commonly used, that this s	inforced was infordeded. (Ind of rationate.)	25
Ad	26		
Advice to users. This function gives the rank of the process in the particular commu- nicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.			27
M	any programs will be written y	with the master-slave model, where one process (such	28 29
as the rank-zero process) will play a supervisory role, and the other processes will			29 30
	- / -	is framework, the two preceding calls are useful for	31
	_	ious processes of a communicator. (End of advice to	32
	ers.)		33
			34
			35
	MM_COMPARE(comm1, com	m2 result)	36
	· ·	,	37
IN	comm1	first communicator (handle)	38
IN	comm2	second communicator (handle)	39
OUT	result	result (integer)	40
			41
int MPT	42		
	-	omm1,MPI_Comm comm2, int *result)	43 ticket 229.2.
	m_compare(comm1, comm2, 1		44 ticket-248T.
	PE(MPI_Comm), INTENT(IN) :		45 46
	EGER, INTENT(OUT) :: res		46
TNT	EGER, OPTIONAL, INTENT(OU	JI) :: lerror	48

1

 $\frac{2}{3}$

1 MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) $\mathbf{2}$ INTEGER COMM1, COMM2, RESULT, IERROR 3 {static int MPI::Comm::Compare(const MPI::Comm& comm1, 4 const MPI::Comm& comm2) (binding deprecated, see Section 15.2) } 56 MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical $\overline{7}$ groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical 8 in constituents and rank order; these communicators differ only by context. MPI_SIMILAR 9 results if the group members of both communicators are the same but the rank order differs. 10 MPI_UNEQUAL results otherwise. 11 126.4.2 Communicator Constructors 13 The following are collective functions that are invoked by all processes in the group or 14groups associated with comm. 1516*Rationale.* Note that there is a chicken-and-egg aspect to MPI in that a communicator 17 is needed to create a new communicator. The base communicator for all MPI com-18 municators is predefined outside of MPI, and is MPI_COMM_WORLD. This model was 19 arrived at after considerable debate, and was chosen to increase "safety" of programs 20written in MPI. (End of rationale.) 2122 The MPI interface provides four communicator construction routines that apply to 23both intracommunicators and intercommunicators. The construction routine 24MPI_INTERCOMM_CREATE (discussed later) applies only to intercommunicators. 25An intracommunicator involves a single group while an intercommunicator involves 26two groups. Where the following discussions address intercommunicator semantics, the 27two groups in an intercommunicator are called the *left* and *right* groups. A process in an 28 intercommunicator is a member of either the left or the right group. From the point of view 29 of that process, the group that the process is a member of is called the *local* group; the 30 other group (relative to that process) is the *remote* group. The left and right group labels 31 give us a way to describe the two groups in an intercommunicator that is not relative to 32 any particular process (as the local and remote groups are). 33 3435 MPI_COMM_DUP(comm, newcomm) 36 IN comm communicator (handle) 37 OUT copy of **comm** (handle) 38 newcomm 39 40int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm) ticket-248T. 41 MPI_Comm_dup(comm, newcomm, ierror) BIND(C) 42TYPE(MPI_Comm), INTENT(IN) :: comm 43 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546MPI_COMM_DUP(COMM, NEWCOMM, IERROR) 47INTEGER COMM, NEWCOMM, IERROR 48

<pre>[NP1::Distgraphcomm vP1::Distgraphcomm::Dup() const(binding deprecated, see Section 15.2) } [MP1::Comm& MP1::Conm::Clone() const = 0(binding deprecated, see Section 15.2) } [MP1::Intracomm& MP1::Intracomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Cartcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Cartcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Cartcomm& MP1::Conment::Clone() const(binding deprecated, see Section 15.2) } [MP1::Cartcomm& MP1::Circomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Distgraphcomm& MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1:COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a duplicate communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is availed even if there are pending point-to-point communications involving the communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is availed even if there are pending point-to-point communicators. (End of advice to users.) Advice to implementors.</pre>	<pre>{MPI::Intracomm MPI::Intracomm::Dup() const(binding deprecated, see Section 15.2) }</pre>	1 2
<pre>[mp1::Cont count mp1::Cont count :Dup() Const(binding deprecated, see Section 15.2) { [mp1::Conmex MP1::Conmex:Clone() const(binding deprecated, see Section 15.2) { [mp1::Comm& MP1::Comm::Clone() const = 0(binding deprecated, see Section 15.2) { [mp1::Comm& MP1::Comm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Intracomm& MP1::Intracomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Cartcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Gartcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Gartcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Gartcomm& MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Distgraphcomm MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Distgraphcomm& MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) { [mp1::Distgraphcomm MP1::Distgraphcomm::Clone() const(binding d</pre>	<pre>{MPI::Intercomm MPI::Intercomm::Dup() const(binding deprecated, see Section 15.2) }</pre>	4
<pre>{MPI::Graphcomm MPI::Graphcomm::Dup() const(binding deprecated, see Section 15.2)</pre>	{MPI::Cartcomm MPI::Cartcomm::Dup() const(binding deprecated, see Section 15.2) }	
<pre>[NP1::Distgraphcomm MP1::Distgraphcomm::Dup() Const(binding deprecated, see Section 15.2) } [MP1::Comm& MP1::Comm::Clone() const = 0(binding deprecated, see Section 15.2) } [MP1::Intracomm& MP1::Intracomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Cartcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Graphcomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Graphcomm& MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Distgraphcomm& MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1:COMM_DUP Duplicates the existing communicator comm with associated key val- ues. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a dupli- cate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the commu- nicator comm. A typical call might involve a MP1_COMM_DUP at the beginning of the parallel call, and an MP1_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.)</pre>	<pre>{MPI::Graphcomm MPI::Graphcomm::Dup() const(binding deprecated, see Section 15.2) }</pre>	8
<pre>{MPI::Comm& MPI::Comm::Clone() const = 0(binding deprecated, see Section 15.2) } {MPI::Intracomm& MPI::Intracomm::Clone() const(binding deprecated, see</pre>		10 11
 [NP1::Intracomm 'NF1::Intracom:::Clone() const(binding deprecated, see Section 15.2) } [MP1::Intercomm& MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Cartcomm& MP1::Graphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Graphcomm& MP1::Graphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Distgraphcomm& MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Distgraphcomm MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1::Distgraphcomm MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } [MP1:COMM_DUP Duplicates the existing communicator comm with associated key value. Section 15.2) } [MP1_COMM_DUP Duplicates the existing communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new communicator with the same group or groups, any copied cached information, but a about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a dupli- cate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the commu- nicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.) 	<pre>{MPI::Comm& MPI::Comm::Clone() const = 0(binding deprecated, see Section 15.2) }</pre>	13
<pre>{MPI::Intercomm& MPI::Intercomm::Clone() const(binding deprecated, see Section 15.2) } {MPI::Cartcomm& MPI::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } {MPI::Graphcomm& MPI::Corector() const(binding deprecated, see Section 15.2) } {MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } {MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } {MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } {MPI:COMM_DUP Duplicates the existing communicator comm with associated key val- ues. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy call- back may take is to delete the attribute from the new communicator. Returns in newcomm a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a dupli- cate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the commu- nicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.)</pre>		14 15
<pre>{MP1::Cartcomm# MP1::Cartcomm::Clone() const(binding deprecated, see Section 15.2) } { MP1::Graphcomm# MP1::Graphcomm::Clone() const(binding deprecated, see Section 15.2) { MP1::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) { MP1.COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy call- back may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a anew communication space that has the same properties as the original communicator. Advice to users. This operation is used to provide a parallel library call with a duplicate communicator space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.) are interval. are interval.</pre>		17
 {MPI::Graphcomm& MPI::Graphcomm::Clone() const(binding deprecated, see Section 15.2) } {MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.) 	<pre>{MPI::Cartcomm& MPI::Cartcomm::Clone() const(binding deprecated, see Section 15.2) }</pre>	19 20 21
 {MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) } MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.) 		22 22 23
 MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion about the C++ bindings for Dup() and Clone(). Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicators. (End of advice to users.) Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(End of advice to implementors.) 		25
Advice to users.This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator.36This includes any attributes (see below), and topologies (see Chapter 7).3738valid even if there are pending point-to-point communications involving the communicator comm.3930303132333434353536363738393939393030313233343535363636373839 <td>ues. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion</td> <td>27 28 29 30 31 32 33</td>	ues. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 636 for further discussion	27 28 29 30 31 32 33
Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(<i>End of advice to implementors.</i>) 43 44 45 46 47	cate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the commu- nicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the	35 36 37 38 39 40
Advice to implementors.One need not actually copy the group information, but only44add a new reference and increment the reference count.Copy on write can be used45for the cached information.(End of advice to implementors.)46	This call applies to both intra- and inter-communicators. (End of advice to users.)	42
	add a new reference and increment the reference count. Copy on write can be used	

CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING 254

12

MPI_COMM_CREATE(comm, group, newcomm)

	2 3	IN	comm	communicator (handle)
	4 5	IN	group	Group, which is a subset of the group of comm (handle)
	6 7	OUT	newcomm	new communicator (handle)
ticket-248T.	8 9	int MPI_C	omm_create(MPI_Comm	<pre>comm, MPI_Group group, MPI_Comm *newcomm)</pre>
	10 11		create(comm, group, MPI_Comm), INTENT(II	newcomm, ierror) BIND(C) N) :: comm
	12	TYPE(MPI_Group), INTENT(IN) :: group
	13 14		MPI_Comm), INTENT(O ER, OPTIONAL, INTEN	
	15 16 17		CREATE(COMM, GROUP, ER COMM, GROUP, NEW	
	18 19	$\{MPI::Int$		mm::Create(const MPI::Group& group) precated, see Section 15.2) }
	20 21 22	$\{MPI::Int\}$		<pre>mm::Create(const MPI::Group& group) precated, see Section 15.2) }</pre>
	22 23 24		,	this function returns a new communicator newcomm with the group argument. No cached information propagates
	24	$\mathrm{from}\;\mathrm{comm}$	to newcomm. Each pro	ocess must call with a group argument that is a subgroup
	26 27	-	-	n; this could be MPI_GROUP_EMPTY. The processes may up argument. If a process calls with a non-empty group
	28		_	nust call the function with the same group as argument,
	29 30		-	ame order. Otherwise the call is erroneous. This implies coss the processes must be disjoint. If the calling process
	31			group argument, then newcomm is a communicator with
	32			he case that a process calls with a group to which it does
	33 34			TY, then MPI_COMM_NULL is returned as newcomm. The called by all processes in the group of comm.
	35	_		
	36 37			processes of comm. It was extended in MPI-2.2 to allow
	38	-	0 1	s in order to allow implementations to eliminate unnec-
	39	essar	y communication that	MPI_COMM_SPLIT would incur when the user already
	40	know	s the membership of th	e disjoint subgroups. (End of rationale.)
	41	Ratic	male The requirement	t that the entire group of comm participate in the call
	42 43		s from the following cor	
	44		It allows the implemen	tation to layor MDL COMM CREATE on top of regular
	45		collective communication	tation to layer MPI_COMM_CREATE on top of regular ons.
	46			afety, in particular in the case where partially overlapping
	47 48		groups are used to crea	

Unofficial Draft for Comment Only

• It permits implementations sometimes to avoid communication related to context creation.

(End of rationale.)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) further to subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

Advice to implementors. When calling MPI_COMM_DUP, all processes call with the same group (the group associated with the communicator). When calling MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups. For both calls, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of newcomm. All processes in the same local group of comm must specify the same value for group, i.e., the same members in the same order. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Rationale. In the case where either the left or right group is empty, a null communicator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (*End of rationale.*)

Example 6.1 The following example illustrates how the first node in the left side of an intercommunicator could be joined with all members on the right side of an intercommunicator to form a new intercommunicator.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 31

```
1
                               INTER-COMMUNICATOR CREATE
2
                       Before
3
4
                              0
                              5
6
                         0
7
                         4
8
                                                                   2
                                   IŌ
9
10
11
                                 1
                                                               ١
                                                             ١
                       After
12
                                  I
13
                                1
14
                              00
15
                                                                  1
16
17
                                                                      6
                                                                 2
18
19
20
21
22
     Figure 6.1: Intercommunicator create using MPI_COMM_CREATE extended to intercom-
23
     municators. The input groups are those in the grey circle.
24
25
              MPI_Comm inter_comm, new_inter_comm;
26
              MPI_Group local_group, group;
27
                         rank = 0; /* rank on left side to include in
              int
28
                                        new inter-comm */
29
30
              /* Construct the original intercommunicator: "inter_comm" */
^{31}
              . . .
32
33
              /* Construct the group of processes to be in new
34
                  intercommunicator */
35
              if (/* I'm on the left side of the intercommunicator */) {
36
                MPI_Comm_group ( inter_comm, &local_group );
37
                MPI_Group_incl ( local_group, 1, &rank, &group );
38
                MPI_Group_free ( &local_group );
39
              }
40
              else
41
                MPI_Comm_group ( inter_comm, &group );
42
43
              MPI_Comm_create ( inter_comm, group, &new_inter_comm );
44
              MPI_Group_free( &group );
45
46
47
48
```

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)	2
11 1	comm	communicator (nanue)	3
IN	color	control of subset assignment (integer)	4
IN	key	control of rank assignment (integer)	5
OUT	newcomm	new communicator (handle)	6
001	newconnin	new communicator (nandle)	_

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

- MPI_Comm_split(comm, color, key, newcomm, ierror) BIND(C)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: color, key
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
 INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all processes that are not members of their group argument provide color = MPI_UNDEFINED. The value of color must be non poretive

The value of color must be non-negative.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra. For intracommunicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful when some processes do not have complete information of the other members in their group, but all processes know (the color of) the group to which they belong. In this case, the MPI implementation discovers the other group members via communication. MPI_COMM_CREATE is useful when all processes have complete information

 24

 31

 $^{9}_{10}$ ticket-248T.

- of the members of their group. In this case, MPI can avoid the extra communication
 required to discover group membership.
- ³ Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that ⁴ any call have no overlap of the resulting communicators (each process is of only one ⁵ color per call). In this way, multiple overlapping communication structures can be ⁶ created. Creative use of the color and key in such splitting operations is encouraged.
- ⁸ Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's ⁹ responsibility to sort processes in ascending order according to this key, and to break ¹⁰ ties in a consistent way. If all the keys are specified in the same way, then all the ¹¹ processes in a given color will have the relative rank order as they did in their parent ¹² group.
- Essentially, making the key value zero for all processes of a given color means that one doesn't really care about the rank-order of the processes in the new communicator.
 (*End of advice to users.*)

17

18

26

27

28

29

30 31

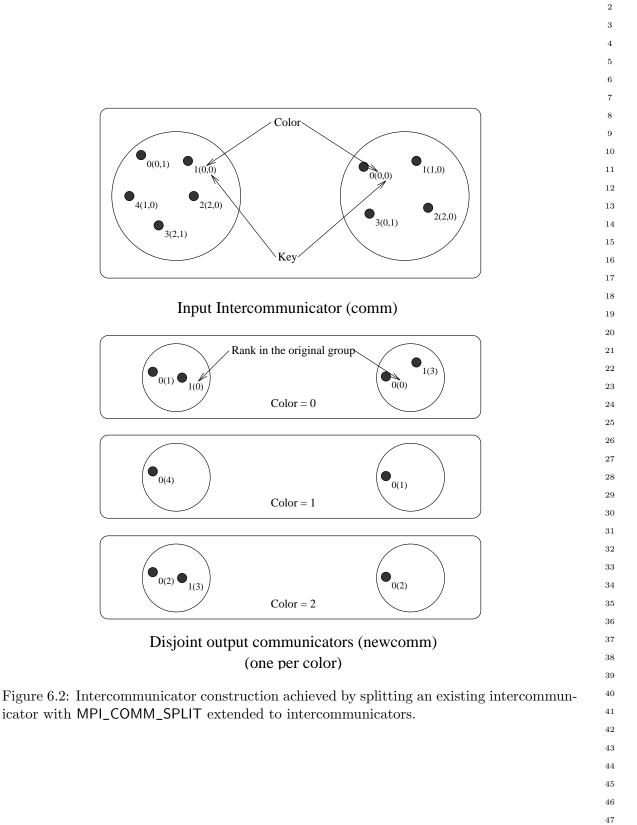
32

33

34

- *Rationale.* color is restricted to be non-negative, so as not to confict with the value assigned to MPI_UNDEFINED. (*End of rationale.*)
- The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.
 - Advice to users. For intercommunicators, MPI_COMM_SPLIT is more general than MPI_COMM_CREATE. A single call to MPI_COMM_SPLIT can create a set of disjoint intercommunicators, while a call to MPI_COMM_CREATE creates only one. (*End of advice to users.*)
 - **Example 6.2** (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
35
             /* Client code */
36
             MPI_Comm multiple_server_comm;
37
             MPI_Comm
                        single_server_comm;
38
              int
                        color, rank, num_servers;
39
40
              /* Create intercommunicator with clients and servers:
41
                 multiple_server_comm */
42
              . . .
43
44
             /* Find out the number of servers available */
45
             MPI_Comm_remote_size ( multiple_server_comm, &num_servers );
46
47
             /* Determine my color */
48
             MPI_Comm_rank ( multiple_server_comm, &rank );
```



```
1
                          color = rank % num_servers;
            2
            3
                          /* Split the intercommunicator */
            4
                          MPI_Comm_split ( multiple_server_comm, color, rank,
            5
                                             &single_server_comm );
            6
                 The following is the corresponding server code:
            7
            8
                          /* Server code */
            9
                          MPI_Comm multiple_client_comm;
            10
                          MPI_Comm single_server_comm;
            11
                          int
                                     rank;
           12
            13
                          /* Create intercommunicator with clients and servers:
           14
                             multiple_client_comm */
            15
                          . . .
            16
            17
                          /* Split the intercommunicator for a single server per group
            18
                              of clients */
            19
                          MPI_Comm_rank ( multiple_client_comm, &rank );
           20
                          MPI_Comm_split ( multiple_client_comm, rank, 0,
           21
                                             &single_server_comm );
           22
           23
                 6.4.3 Communicator Destructors
           24
           25
           26
                 MPI_COMM_FREE(comm)
           27
                   INOUT
                                                         communicator to be destroyed (handle)
                            comm
           28
           29
            30
                 int MPI_Comm_free(MPI_Comm *comm)
ticket-248T. 31
                 MPI_Comm_free(comm, ierror) BIND(C)
           32
                     TYPE(MPI_Comm), INTENT(INOUT) ::
                                                           COMM
           33
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           34
           35
                 MPI_COMM_FREE(COMM, IERROR)
           36
                     INTEGER COMM, IERROR
           37
                 {void MPI::Comm::Free() (binding deprecated, see Section 15.2) }
           38
           39
                     This collective operation marks the communication object for deallocation. The handle
           40
                 is set to MPI_COMM_NULL. Any pending operations that use this communicator will com-
           41
                 plete normally; the object is actually deallocated only if there are no other active references
           42
                 to it. This call applies to intra- and inter-communicators. The delete callback functions for
           43
                 all cached attributes (see Section 6.7) are called in arbitrary order.
           44
           45
                      Advice to implementors. A reference-count mechanism may be used: the reference
            46
                      count is incremented by each call to MPI_COMM_DUP, and decremented by each call
            47
                      to MPI_COMM_FREE. The object is ultimately deallocated when the count reaches
            48
                      zero.
```

Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (End of advice to implementors.)

6.5 Motivating Examples

```
6.5.1 Current Practice #1
```

```
Example #1a:
```

```
int main(int argc, char **argv)
{
  int me, size;
  . . .
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  (void)printf ("Process %d size %d\n", me, size);
  . . .
  MPI_Finalize();
}
```

Example #1a is a do-nothing program that initializes itself legally, and refers to the "all" communicator, and prints a message. It terminates itself legally too. This example does not imply that MPI supports printf-like communication itself. Example #1b (supposing that size is even):

```
int main(int argc, char **argv)
                                                                                 27
                                                                                 28
{
                                                                                 29
   int me, size;
                                                                                 30
   int SOME_TAG = 0;
   . . .
   MPI_Init(&argc, &argv);
                                                                                 33
                                                                                 34
   MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                           /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
                                                                                 35
                                                                                 36
                                                                                 37
   if((me % 2) == 0)
   {
                                                                                 38
                                                                                 39
      /* send unless highest-numbered process */
      if((me + 1) < size)
                                                                                 41
         MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
                                                                                 42
   }
   else
                                                                                 43
                                                                                 44
      MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
   MPI_Finalize();
}
```

1

2

3

4 5

6 7

8

9 10

11

12

13

14

15

16

17 18

19

20

21

22

25

26

31

32

40

454647

48

₂₃ ticket 182.

 $_{24}$ ticket 182.

1 Example #1b schematically illustrates message exchanges between "even" and "odd" pro- $\mathbf{2}$ cesses in the "all" communicator. 3 4 6.5.2 Current Practice #2 5int main(int argc, char **argv) 6 { 7 int me, count; 8 void *data; 9 . . . 10 11 MPI_Init(&argc, &argv); 12MPI_Comm_rank(MPI_COMM_WORLD, &me); 13 14if(me == 0)15{ 16/* get input, create buffer ''data'' */ 17. . . 18 } 1920MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD); 2122 . . . 23 24 MPI_Finalize(); 25} 2627This example illustrates the use of a collective communication. 2829 6.5.3 (Approximate) Current Practice #3 30 31 int main(int argc, char **argv) 32 { 33int me, count, count2; 34void *send_buf, *recv_buf, *send_buf2, *recv_buf2; 35 MPI_Group MPI_GROUP_WORLD, grprem; 36 MPI_Comm commslave; 37 static int ranks[] = {0}; 38. . . 39 MPI_Init(&argc, &argv); 40MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD); 41 MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */ 4243 MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem); /* local */ 44MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave); 4546if(me != 0)47{ 48 /* compute on slave */

```
...
MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commslave);
...
MPI_Comm_free(&commslave);
}
/* zero falls through immediately to this reduce, others do later... */
MPI_Reduce(send_buf2, recv_buf2, count2,
MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Group_free(&MPI_GROUP_WORLD);
MPI_Group_free(&grprem);
MPI_Finalize();
}
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commslave) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

6.5.4 Example #4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
29
#define TAG_ARBITRARY 12345
                                                                                   30
#define SOME_COUNT
                           50
                                                                                   31
                                                                                   32
int main(int argc, char **argv)
                                                                                   33
{
                                                                                   34
  int me;
  MPI_Request request[2];
                                                                                   35
                                                                                   36
  MPI_Status status[2];
                                                                                   37
  MPI_Group MPI_GROUP_WORLD, subgroup;
                                                                                   38
  int ranks[] = {2, 4, 6, 8};
                                                                                   39
  MPI_Comm the_comm;
                                                                                   40
  . . .
                                                                                   41
  MPI_Init(&argc, &argv);
                                                                                   42
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
                                                                                   43
                                                                                   44
  MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
  MPI_Group_rank(subgroup, &me);
                                                                                   45
                                        /* local */
                                                                                   46
                                                                                   47
  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
                                                                                   48
```

1

2

4

5 6

7

8

9 10

11

12

13 14

15

16

17

18

19

20

21

22 23

 $\frac{24}{25}$

26

27

```
1
           if(me != MPI_UNDEFINED)
\mathbf{2}
           {
3
               MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
4
                                    the_comm, request);
5
               MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
6
                                    the_comm, request+1);
7
               for(i = 0; i < SOME_COUNT; i++)</pre>
8
                  MPI_Reduce(..., the_comm);
9
               MPI_Waitall(2, request, status);
10
11
               MPI_Comm_free(&the_comm);
           }
12
13
14
           MPI_Group_free(&MPI_GROUP_WORLD);
15
           MPI_Group_free(&subgroup);
16
           MPI_Finalize();
17
        }
18
19
            Library Example \#1
     6.5.5
20
     The main program:
21
22
         int main(int argc, char **argv)
23
         {
24
           int done = 0;
25
           user_lib_t *libh_a, *libh_b;
26
           void *dataset1, *dataset2;
27
           . . .
28
           MPI_Init(&argc, &argv);
29
           . . .
30
           init_user_lib(MPI_COMM_WORLD, &libh_a);
^{31}
           init_user_lib(MPI_COMM_WORLD, &libh_b);
32
           . . .
33
           user_start_op(libh_a, dataset1);
34
           user_start_op(libh_b, dataset2);
35
           . . .
36
           while(!done)
37
           {
38
              /* work */
39
              . . .
40
              MPI_Reduce(..., MPI_COMM_WORLD);
41
              . . .
42
              /* see if done */
43
              . . .
44
           }
45
           user_end_op(libh_a);
46
           user_end_op(libh_b);
47
```

```
1
     uninit_user_lib(libh_a);
                                                                                          \mathbf{2}
     uninit_user_lib(libh_b);
                                                                                          3
     MPI_Finalize();
   }
                                                                                          4
                                                                                          5
The user library initialization code:
                                                                                          6
                                                                                          7
   void init_user_lib(MPI_Comm comm, user_lib_t **handle)
                                                                                          8
   {
                                                                                          9
     user_lib_t *save;
                                                                                          10
                                                                                          11
     user_lib_initsave(&save); /* local */
                                                                                          12
     MPI_Comm_dup(comm, &(save -> comm));
                                                                                          13
                                                                                         14
     /* other inits */
                                                                                          15
     . . .
                                                                                          16
                                                                                          17
     *handle = save;
                                                                                          18
   }
                                                                                          19
                                                                                         20
User start-up code:
                                                                                         21
   void user_start_op(user_lib_t *handle, void *data)
                                                                                         22
   {
                                                                                         23
     MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
                                                                                         ^{24}
     MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
                                                                                         25
   }
                                                                                          26
                                                                                         27
User communication clean-up code:
                                                                                         28
   void user_end_op(user_lib_t *handle)
                                                                                         29
                                                                                         30
   {
                                                                                          ^{31}
     MPI_Status status;
     MPI_Wait(& handle -> isend_handle, &status);
                                                                                          32
                                                                                         33
     MPI_Wait(& handle -> irecv_handle, &status);
   }
                                                                                         34
                                                                                         35
User object clean-up code:
                                                                                         36
                                                                                         37
   void uninit_user_lib(user_lib_t *handle)
                                                                                         38
   ſ
                                                                                         39
     MPI_Comm_free(&(handle -> comm));
                                                                                          40
     free(handle);
                                                                                          41
   }
                                                                                         42
                                                                                         43
6.5.6 Library Example #2
                                                                                         44
The main program:
                                                                                          45
                                                                                          46
   int main(int argc, char **argv)
                                                                                          47
   {
```

```
1
          int ma, mb;
\mathbf{2}
          MPI_Group MPI_GROUP_WORLD, group_a, group_b;
3
          MPI_Comm comm_a, comm_b;
4
5
          static int list_a[] = {0, 1};
6
     #if defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
7
          static int list_b[] = {0, 2, 3};
8
     #else/* EXAMPLE_2A */
9
          static int list_b[] = \{0, 2\};
10
     #endif
11
          int size_list_a = sizeof(list_a)/sizeof(int);
12
          int size_list_b = sizeof(list_b)/sizeof(int);
13
14
15
          MPI_Init(&argc, &argv);
16
          MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
17
18
          MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
19
          MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);
20
21
          MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
22
          MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
23
^{24}
          if(comm_a != MPI_COMM_NULL)
25
             MPI_Comm_rank(comm_a, &ma);
26
          if(comm_b != MPI_COMM_NULL)
27
             MPI_Comm_rank(comm_b, &mb);
28
29
          if(comm_a != MPI_COMM_NULL)
30
             lib_call(comm_a);
31
32
          if(comm_b != MPI_COMM_NULL)
33
          ſ
34
            lib_call(comm_b);
35
            lib_call(comm_b);
36
          }
37
38
          if(comm_a != MPI_COMM_NULL)
39
            MPI_Comm_free(&comm_a);
40
          if(comm_b != MPI_COMM_NULL)
41
            MPI_Comm_free(&comm_b);
42
          MPI_Group_free(&group_a);
43
          MPI_Group_free(&group_b);
44
          MPI_Group_free(&MPI_GROUP_WORLD);
45
          MPI_Finalize();
46
        }
47
     The library:
48
```

```
void lib_call(MPI_Comm comm)
   {
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
        }
     else
     {
       /* work */
       MPI_Send(..., 0, ARBITRARY_TAG, comm);
       . . . .
     }
#ifdef EXAMPLE_2C
     /* include (resp, exclude) for safety (resp, no safety): */
     MPI_Barrier(comm);
#endif
   }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no backmasking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [58]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that backmasking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-topoint operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 6.9.

- 45 46
- 40
- 48

1

2

3

4

5

6

78

9 10

11

12 13

14 15

16

17

18

19 20

21

22 23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

3

4

5

29

30

 31

32

33

34

35

36

37

38 39

40

41

42

43

44 45

46

47 48

6.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct 10 modules and processes within different modules communicate with one another in a pipeline 11 or a more general module graph. In these applications, the most natural way for a process 12to specify a target process is by the rank of the target process within the target group. In 13 applications that contain internal user-level servers, each server may be a process group that 14provides services to one or more clients, and each client may be a process group that uses 15the services of one or more servers. It is again most natural to specify the target process 16by rank within the target group in these applications. This type of communication is called 17"inter-communication" and the communicator used is called an "inter-communicator," as 18 introduced earlier. 19

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint.

Advice to users. The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of

MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (*End of advice to users.*)

- Here is a summary of the properties of inter-communication and inter-communicators:
- The syntax of point-to-point and collective communication is the same for both interand intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group send_context receive_context source

For inter-communicators, **group** describes the remote group, and **source** is the rank of the process in the local group. For intra-communicators, **group** is the communicator group (remote=local), **source** is the rank of the process in this group, and **send context** and **receive context** are identical. A group can be represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process \mathbf{P} in group \mathcal{P} , which has an intercommunicator $\mathbf{C}_{\mathcal{P}}$, and a process \mathbf{Q} in group \mathcal{Q} , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $C_{\mathcal{P}}$.group describes the group \mathcal{Q} and $C_{\mathcal{Q}}$.group describes the group \mathcal{P} .
- $C_{\mathcal{P}}$.send_context = $C_{\mathcal{Q}}$.receive_context and the context is unique in \mathcal{Q} ; $C_{\mathcal{P}}$.receive_context = $C_{\mathcal{Q}}$.send_context and this context is unique in \mathcal{P} .
- $C_{\mathcal{P}}$.source is rank of P in \mathcal{P} and $C_{\mathcal{Q}}$.source is rank of Q in \mathcal{Q} .

Assume that \mathbf{P} sends a message to \mathbf{Q} using the inter-communicator. Then \mathbf{P} uses the **group** table to find the absolute address of \mathbf{Q} ; **source** and **send_context** are appended to the message.

Assume that \mathbf{Q} posts a receive with an explicit source argument using the intercommunicator. Then \mathbf{Q} matches **receive_context** to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (*End of advice to implementors.*)

6.6.1 Inter-communicator Accessors

MPI_COMM_TEST_INTER(comm, flag)

IN	comm	communicator (handle)	4
OUT	flag	(logical)	4

Unofficial Draft for Comment Only

```
1
                 int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
ticket-248T. 2
                 MPI_Comm_test_inter(comm, flag, ierror) BIND(C)
            3
                     TYPE(MPI_Comm), INTENT(IN) :: comm
            4
                     LOGICAL, INTENT(OUT) :: flag
            5
                      INTEGER, OPTIONAL, INTENT(OUT) ::
                                                             ierror
            6
            \overline{7}
                 MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
            8
                      INTEGER COMM, IERROR
            9
                     LOGICAL FLAG
            10
                 {bool MPI::Comm::Is_inter() const(binding deprecated, see Section 15.2) }
            11
           12
                 This local routine allows the calling process to determine if a communicator is an inter-
            13
                 communicator or an intra-communicator. It returns true if it is an inter-communicator,
           14
                 otherwise false.
            15
                     When an inter-communicator is used as an input argument to the communicator ac-
            16
                 cessors described above under intra-communication, the following table describes behavior.
            17
            18
                           MPI_COMM_SIZE
                                                  returns the size of the local group.
            19
                           MPI_COMM_GROUP
                                                  returns the local group.
           20
                           MPI_COMM_RANK
                                                  returns the rank in the local group
           21
           22
           23
                       Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)
           ^{24}
            25
                 Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both
           26
                 communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results.
           27
                 Both corresponding local and remote groups must compare correctly to get the results
           28
                 MPI_CONGRUENT and MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result
           29
                 because either the local or remote groups were similar but not identical.
           30
                     The following accessors provide consistent access to the remote group of an inter-
           ^{31}
                 communicator:
            32
                     The following are all local operations.
           33
           34
                 MPI_COMM_REMOTE_SIZE(comm, size)
           35
           36
                   IN
                                                         inter-communicator (handle)
                             comm
           37
                   OUT
                                                         number of processes in the remote group of comm
                             size
            38
                                                         (integer)
            39
            40
                 int MPI_Comm_remote_size(MPI_Comm comm, int *size)
           41
ticket-248T.
            42
                 MPI_Comm_remote_size(comm, size, ierror) BIND(C)
           43
                     TYPE(MPI_Comm), INTENT(IN) ::
                                                         comm
           44
                     INTEGER, INTENT(OUT) :: size
            45
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            46
                 MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
            47
                      INTEGER COMM, SIZE, IERROR
            48
```

```
1
{int MPI::Intercomm::Get_remote_size() const(binding deprecated, see Section 15.2)
                                                                                         \mathbf{2}
               ł
                                                                                         3
                                                                                         4
                                                                                         5
MPI_COMM_REMOTE_GROUP(comm, group)
                                                                                         6
  IN
                                       inter-communicator (handle)
           comm
                                                                                         7
  OUT
           group
                                       remote group corresponding to comm (handle)
                                                                                         8
                                                                                        9
                                                                                        10
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
                                                                                        11 ticket-248T.
MPI_Comm_remote_group(comm, group, ierror) BIND(C)
                                                                                        12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        13
    TYPE(MPI_Group), INTENT(OUT) :: group
                                                                                        14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        15
                                                                                        16
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
                                                                                        17
    INTEGER COMM, GROUP, IERROR
                                                                                        18
{MPI::Group MPI::Intercomm::Get_remote_group() const/binding deprecated, see
                                                                                        19
               Section 15.2 }
                                                                                        20
                                                                                        21
```

Rationale. Symmetric access to both the local and remote groups of an intercommunicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE have been provided. (*End of rationale.*)

6.6.2 Inter-communicator Operations

This section introduces four blocking inter-communicator operations. MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multithreaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.)

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator 39 from two existing intra-communicators, in the following situation: At least one selected 40 member from each group (the "group leader") has the ability to communicate with the 41 selected member from the other group; that is, a "peer" communicator exists to which both 42 leaders belong, and each leader knows the rank of the other leader in this peer communicator. 43 Furthermore, members of each group know the rank of their leader. 44

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

47 48

45

46

22

23

 24

25 26

27

28

29

30

31

32

33

34

35

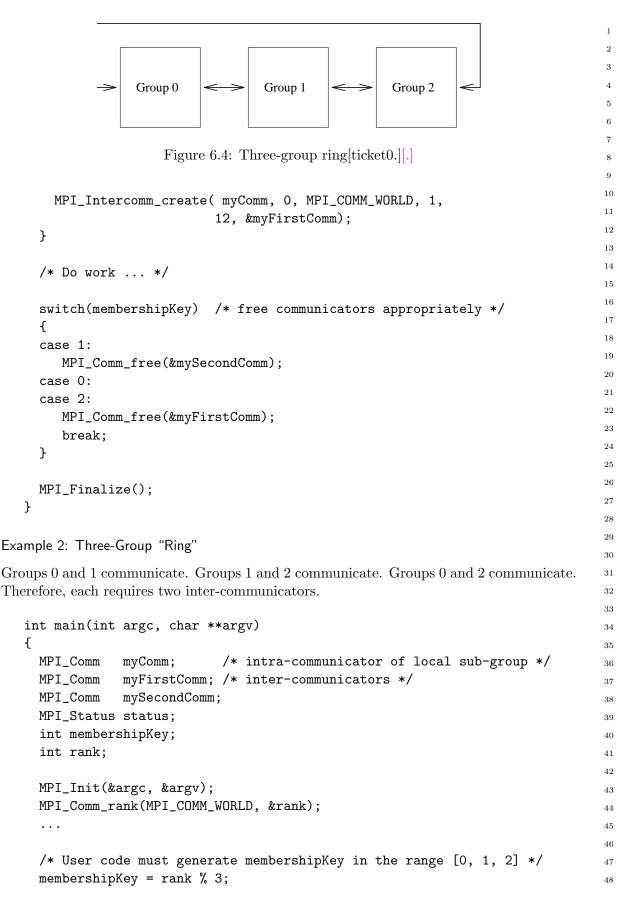
36

37

```
1
                     In standard MPI implementations (with static process allocation at initialization), the
            \mathbf{2}
                 MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this
            3
                 peer communicator. For applications that have used spawn or join, it may be necessary to
            4
                 first create an intracommunicator to be used as peer.
            \mathbf{5}
                      The application topology functions described in Chapter 7 do not apply to inter-
            6
                 communicators. Users that require this capability should utilize
            7
                 MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or carte-
            8
                 sian topology capabilities to that intra-communicator, creating an appropriate topology-
            9
                 oriented intra-communicator. Alternatively, it may be reasonable to devise one's own ap-
            10
                 plication topology mechanisms for this case, without loss of generality.
            11
            12
                 MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag,
            13
                                 newintercomm)
            14
            15
                   IN
                             local_comm
                                                         local intra-communicator (handle)
            16
                   IN
                             local_leader
                                                          rank of local group leader in local_comm (integer)
            17
                   IN
                             peer_comm
                                                          "peer" communicator; significant only at the
            18
                                                         local_leader (handle)
            19
            20
                   IN
                             remote_leader
                                                         rank of remote group leader in peer_comm; significant
            21
                                                          only at the local_leader (integer)
            22
                   IN
                                                          "safe" tag (integer)
                             tag
            23
                   OUT
                             newintercomm
                                                         new inter-communicator (handle)
            ^{24}
            25
            26
                 int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
                                MPI_Comm peer_comm, int remote_leader, int tag,
            27
                                MPI_Comm *newintercomm)
            28
ticket-248T.
           29
                 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
            30
                                tag, newintercomm, ierror) BIND(C)
            ^{31}
                      TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
            32
                      INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
            33
                      TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
            34
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           35
            36
                 MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
           37
                                TAG, NEWINTERCOMM, IERROR)
            38
                      INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
            39
                      NEWINTERCOMM, IERROR
            40
                 {MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader, const
            41
                                MPI::Comm& peer_comm, int remote_leader, int tag) const(binding
            42
                                 deprecated, see Section 15.2 }
            43
            ^{44}
                 This call creates an inter-communicator. It is collective over the union of the local and
            45
                 remote groups. Processes should provide identical local_comm and local_leader arguments
            46
                 within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.
            47
                     This call uses point-to-point communication with communicator
            48
                 peer_comm, and with tag tag between the leaders. Thus, care must be taken that there be
```

no pe	1				
	Advise to weeks We read	ammend using a dedicated near communicator, such as a	2 3		
Advice to users. We recommend using a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators. (End of					
	advice to users.)				
			6		
			7		
			8		
		rcomm, high, newintracomm)	9		
IN	intercomm	Inter-Communicator (handle)	10		
IN	high	(logical)	11		
OU	T newintracomm	new intra-communicator (handle)	12		
			13		
int	MPT Intercomm merge(MPT	_Comm intercomm, int high,	14 15		
	MPI_Comm *newi	-	16		
			$^{10}_{17}$ ticket-248T.		
		mm, high, newintracomm, ierror) BIND(C)	18		
	TYPE(MPI_Comm), INTENT(LOGICAL, INTENT(IN) ::		19		
	TYPE(MPI_Comm), INTENT(•	20		
	INTEGER, OPTIONAL, INTE		21		
			22		
		MM, HIGH, NEWINTRACOMM, IERROR)	²³ ticket250-V.		
	INTEGER INTERCOMM, NEWI	NTRACOMM, IERROR	$_{24}$ ticket250-V.		
LOGICAL HIGH			25		
$\{MPI$::Intracomm MPI::Interco	omm::Merge(bool high) const(binding deprecated, see	26 27		
	Section 15.2 }				
This	function creates an intra-c	ommunicator from the union of the two groups that are	28 29		
		processes should provide the same high value within each	30		
		n one group provided the value $high = false$ and processes	31		
		value $high = true$ then the union orders the "low" group	32		
befor	e the "high" group. If all p	processes provided the same high argument then the order	33		
of th	e union is arbitrary. This c	call is blocking and collective within the union of the two	34		
group			35		
		new intercommunicator in each process is inherited from	36		
		tes the local group. Note that this can result in different	37		
proce	esses in the same communication	ator having different error handlers.	38		
	Advice to implementors.	The implementation of MPI_INTERCOMM_MERGE,	39 40		
	-	/PI_COMM_DUP are similar to the implementation of	40		
		E, except that contexts private to the input inter-com-	42		
		mmunication between group leaders rather than contexts	43		
		tor. (End of advice to implementors.)	44		
			45		
			46		
			47		

```
1
2
3
                      Group 0
                                            Group 1
                                                                 Group 2
                                  \leftarrow
                                     \rightarrow
                                                       \leftarrow
                                                           \rightarrow
4
5
6
7
                           Figure 6.3: Three-group pipeline[ticket0.][.]
8
9
     6.6.3 Inter-Communication Examples
10
11
     Example 1: Three-Group "Pipeline"
12
     Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires
13
     one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1
14
     inter-communicator.
15
16
         int main(int argc, char **argv)
17
         {
18
                       myComm;
           MPI_Comm
                                       /* intra-communicator of local sub-group */
19
                       myFirstComm; /* inter-communicator */
           MPI_Comm
20
                       mySecondComm; /* second inter-communicator (group 1 only) */
           MPI_Comm
21
           int membershipKey;
22
           int rank;
23
24
           MPI_Init(&argc, &argv);
25
           MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26
27
           /* User code must generate membershipKey in the range [0, 1, 2] */
28
           membershipKey = rank % 3;
29
30
           /* Build intra-communicator for local sub-group */
31
           MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
32
33
           /* Build inter-communicators. Tags are hard-coded. */
34
           if (membershipKey == 0)
35
           {
                                    /* Group 0 communicates with group 1. */
36
             MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
37
                                     1, &myFirstComm);
38
           }
39
           else if (membershipKey == 1)
40
                            /* Group 1 communicates with groups 0 and 2. */
           {
41
             MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
42
                                     1, &myFirstComm);
43
             MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
44
                                     12, &mySecondComm);
45
           }
46
           else if (membershipKey == 2)
47
                                    /* Group 2 communicates with group 1. */
           {
48
```



```
1
2
          /* Build intra-communicator for local sub-group */
3
          MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
4
5
          /* Build inter-communicators. Tags are hard-coded. */
6
          if (membershipKey == 0)
7
          {
                         /* Group 0 communicates with groups 1 and 2. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
8
9
                                   1, &myFirstComm);
10
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
11
                                   2, &mySecondComm);
12
          }
13
          else if (membershipKey == 1)
14
          {
                     /* Group 1 communicates with groups 0 and 2. */
15
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
16
                                   1, &myFirstComm);
17
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
18
                                   12, &mySecondComm);
19
          }
20
          else if (membershipKey == 2)
21
          {
                    /* Group 2 communicates with groups 0 and 1. */
22
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
23
                                   2, &myFirstComm);
24
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
25
                                   12, &mySecondComm);
26
          }
27
28
          /* Do some work ... */
29
30
          /* Then free communicators before terminating... */
31
          MPI_Comm_free(&myFirstComm);
32
          MPI_Comm_free(&mySecondComm);
33
          MPI_Comm_free(&myComm);
34
          MPI_Finalize();
35
        }
36
```

6.7 Caching

37

38 39

40

41

42

43

44 45

46 47

48

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attributing-caching mechanism. (*End of advice to users.*)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. To overcome this problem with attribute caching on communicators, functions are also given for this case. The functions to cache on datatypes and windows also address this issue. For a general discussion of the address size problem, see Section 16.3.6.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

6.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI_COMM_DUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Advice to users. Attributes in C are of type void *. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (*End of advice to users.*)

Advice to implementors. Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, and datatype. Accessor functions include the following:

 $\mathbf{2}$

 $\overline{7}$

 24

 $\frac{44}{45}$

	1 2 3	tion	· · · · · ·	entify an attribute); the user specifies "callback" func- ne application when the communicator is destroyed or		
	4 5	• sto	re and retrieve the value of	an attribute;		
	6 7 8 9 10	in 1 pea	response to explicit applicat	ing and callback functions are only called synchronously, ion requests. This avoid problems that result from re- and system space. (This synchronous calling rule is a		
	11 12 13	im		nder control of MPI. This allows MPI to optimize its ts. It also avoids conflict between independent modules ne communicators.		
	14 15 16 17 18	cac bac cor	hing facility to be implement the interface, some form of ta numicators. In contrast, t	isting of just a callback facility, would allow the entire need by portable code. However, with the minimal call- ble searching is implied by the need to handle arbitrary he more complete interface defined here permits rapid e use of pointers in communicators (to find the attribute		
	19 20 21 22	tab effi	le) and cleverly chosen key ciency "hit" inherent in the	values (to retrieve individual attributes). In light of the minimal interface, the more complete interface defined and of advice to implementors.)		
	23 24 25	-	C C	related to caching. They are all process local.		
	26 27 28		communicators s for caching on communica	itors are:		
	29 30 31	MPI_CO	MM_CREATE_KEYVAL(con extra_state)	nm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,		
	32	IN	comm_copy_attr_fn	copy callback function for $comm_keyval\xspace$ (function)		
	33	IN	comm_delete_attr_fn	delete callback function for comm_keyval (function)		
	34	OUT	comm_keyval	key value for future access (integer)		
	35 36	IN	extra_state	extra state for callback functions		
ticket-248T.	37 38 39 40	<pre>int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,</pre>				
ticket-2461.	41	PRO PRO INT	extra_state, ierr CEDURE(MPI_Comm_copy_at CEDURE(MPI_Comm_delete_ EGER, INTENT(OUT) :: c	<pre>tr_function) :: comm_copy_attr_fn attr_function) :: comm_delete_attr_fn omm_keyval</pre>		
	40 47 48		EGER(KIND=MPI_ADDRESS_K EGER, OPTIONAL, INTENT(IND), INTENT(IN) :: extra_state OUT) :: ierror		

```
1
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
                                                                                        2
               EXTRA_STATE, IERROR)
                                                                                        3
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
                                                                                        \mathbf{4}
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                        5
                                                                                        6
{static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function*
               comm_copy_attr_fn,
               MPI::Comm::Delete_attr_function* comm_delete_attr_fn,
                                                                                        9
               void* extra_state) (binding deprecated, see Section 15.2) }
                                                                                        10
                                                                                        11
    Generates a new attribute key. Keys are locally unique in a process, and opaque to
user, though they are explicitly stored in integers. Once allocated, the key value can be
                                                                                        12
                                                                                        13
used to associate attributes and access them on any locally defined communicator.
                                                                                        14
    This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding
                                                                                        15
is identical. The Fortran binding differs in that extra_state is an address-sized integer.
                                                                                        16
Also, the copy and delete callback functions have Fortran bindings that are consistent with
                                                                                        17
address-sized attributes.
                                                                                        18
The C callback functions are:
                                                                                        19
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                                                                        20
               void *extra_state, void *attribute_val_in,
                                                                                        21
               void *attribute_val_out, int *flag);
                                                                                        22
                                                                                        23
and
                                                                                        ^{24}
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                                                                        25
               void *attribute_val, void *extra_state);
                                                                                        26
which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
                                                                                        27
                                                                                        28
With the mpi_f08 module, the Fortran callback functions are:
                                                                                          ticket230-B.
                                                                                        29
                                                                                          ticket-248T.
                                                                                        30
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
                                                                                        ^{31}
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                        32
       TYPE(MPI_Comm) :: oldcomm
                                                                                        33
       INTEGER :: comm_keyval, ierror
                                                                                        34
       INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                        35
       attribute_val_out
                                                                                        36
       LOGICAL :: flag
                                                                                        37
                                                                                        38
                                                                                          ticket230-B.
and
                                                                                        30
                                                                                          ticket-248T.
ABSTRACT INTERFACE
                                                                                        40
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
                                                                                        41
  attribute_val, extra_state, ierror) BIND(C)
                                                                                        42
      TYPE(MPI_Comm) :: comm
                                                                                        43
       INTEGER :: comm_keyval, ierror
                                                                                        44
       INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                        45
                                                                                        46
                                                                                        ticket230-B.
The With the mpi module and mpif.h, the Fortran callback functions are:
SUBROUTINE COMM_COPY_ATTR_[FN] FUNCTION (OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                                                                                        48 ticket250-V.
```

1 2 3 4 5 6 ticket250-V. 8 9 10	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDCOMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG and SUBROUTINE COMM_DELETE_ATTR_[FN]FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR
11 12	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE The C++ callbacks are:
13 14 15 16 17	<pre>{typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,</pre>
18 19 20 21 22	<pre>and {typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,</pre>
23 24 25 26 ticket322. 27 ticket322. 28 29	The comm_copy_attr_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. comm_copy_attr_fn should be of type MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns $flag = 0$ or .FALSE., then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case
$30 \\ 31 \\ 32 \\ ticket 322. 33 \\ 34 \\ 35 \\ ticket 322. 36 \\ 37 \\ 37 \\ 37 \\ 30 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 32 \\ 31 \\ 31$	MPI_COMM_DUP will fail). The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C, C++, or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 or .FALSE. (depending on whether the keyval was created with a C/C++ or Fortran bind- ing to MPI_COMM_CREATE_KEYVAL) and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple- minded copy function that sets flag = 1 or .TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These replace the MPI-1 predefined callbacks
38 39 40 41 42 43 44	MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated. Advice to users. Even though both formal arguments attribute_val_in and attribute_val_out are of type void *, their usage differs. The C copy function is passed by MPI in attribute_val_in the value of the attribute, and in attribute_val_out the address of the attribute, so as to allow the function to return the (new) attribute value. The use of type void * for both is to avoid messy type casts.
45 46 47 48	A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count

mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (*End of advice to users.*)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows. The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR. comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C, C++, or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

Advice to implementors. To be able to use the predefined C functions MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN as comm_copy_attr_fn argument and/or MPI_COMM_NULL_DELETE_FN as the comm_delete_attr_fn argument in a call to the C++ routine MPI::Comm::Create_keyval, this routine may be overloaded with 3 additional routines that accept the C functions as the first, the second, or both input arguments (instead of an argument that matches the C++ prototype). (End of advice to implementors.)

Advice to users. If a user wants to write a "wrapper" routine that internally calls MPI::Comm::Create_keyval and comm_copy_attr_fn and/or comm_delete_attr_fn are arguments of this wrapper routine, and if this wrapper routine should be callable with both user-defined C++ copy and delete functions and with the predefined C functions, then the same overloading as described above in the advice to implementors may be necessary. (*End of advice to users.*)

³⁸ ₃₉ ticket230-B.

The predefined Fortran functions 40 Advice to implementors. MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and 41 42MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and the mpi_f08 module with the same name, but with different interfaces. Each function 4344can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other 4546routine within mpi_f08 declared with CONTAINS. These routines have different link 47names, which are also different to the link names used for the routines used in C and 48 C++. (End of advice to implementors.)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28 29

30

 31

32

33

34

35

36

		282 C	HAPTER 6.	GROUPS, CC	ONTEXTS, COMMUNICATORS, AND CACHING	
	1 2 3 4 5 6 7	MPI_ MPI_ that u	COMM_NULI uses the mpi_1 if.h, and vic	L_COPY_FN,M L_DELETE_FN £08 module to a	luding the predefined Fortran functions MPI_COMM_DUP_FN, and should not be passed from one application routine nother application routine that uses the mpi module of the advice to users on page 703. (<i>End of advice to</i>	
	8 9					
	10	MPI_COMI	M_FREE_KEY	/VAL(comm_key	/val)	
	11	INOUT	comm_keyva	d	key value (integer)	
	12			- / .		
ticket-248T.	13 • 14	int MPI_C	omm_free_ke	yval(int *com	m_keyval)	
			· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	ierror) BIND(C)	
	16			INOUT) :: cc	•	
	17		-	L, INTENT(OUT		
	18 19			(COMM_KEYVAL,	IERROR)	
	20	INTEG.	ER CUMM_KEY	VAL, IERROR		
	21	$\{ \texttt{static } v \}$			al(int& comm_keyval)(binding deprecated, see	
	22		Section	15.2) }		
	23			-	is function sets the value of keyval to	
	24 25	MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators				
	26	on the process) to the key have been freed. These references need to be explicitly freed by the				
	27	program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,				
	28	or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed				
	29	communicator.				
	30 31	This call is identical to the MPI-1 call MPI_KEYVAL_FREE but is needed to match the				
	32	new commu	inicator-speci	fic creation fund	etion. The use of MPI_KEYVAL_FREE is deprecated.	
	33					
	34	MPI_COM	M_SET_ATTF	R(comm, comm ₋	_keyval, attribute_val)	
	35 36	INOUT	comm		communicator from which attribute will be attached (handle)	
	37	IN	comm_keyva	d	key value (integer)	
	38 39	IN	attribute_val		attribute value	
	40					
	41	int MPI_C	omm_set_att:	r(MPI_Comm co	mm, int comm_keyval, void *attribute_val)	
ticket-248T.		MPT Comm	set attr(co	mm comm kevv	al, attribute_val, ierror) BIND(C)	
	43			INTENT(IN) ::		
	44 45	INTEG	ER, INTENT(IN) :: comm_	keyval	
	46), INTENT(IN) :: attribute_val	
	47	INTEG	ER, OPTIONA	L, INTENT(OUT) :: ierror	
	48	MPI_COMM_	SET_ATTR(CO	MM, COMM_KEYV	AL, ATTRIBUTE_VAL, IERROR)	

INTE	GER COMM, COMM_KEYVAL, IE	RROR	1
INTE) ATTRIBUTE_VAL	2	
{void MP	3		
(****	<pre>m_keyval, const void* attribute_val) ed, see Section 15.2) }</pre>	4	
	5		
	_	attribute value attribute_val for subsequent retrieval	6 7
		lue is already present, then the outcome is as if	8
		called to delete the previous value (and the callback	9
		uted), and a new value was next stored. The call	10
	-	ue keyval; in particular MPI_KEYVAL_INVALID is an the comm_delete_attr_fn function returned an error	11
	than MPI_SUCCESS.	the comm_delete_attr_m function returned an error	12
		_PUT, whose use is deprecated. The C binding is	13
	_	that attribute_val is an address-sized integer.	14
identificai.	The Forman binding differs in	i that attribute_val is an address-sized integer.	15
			16
MPI_COM	IM_GET_ATTR(comm, comm_	_keyval, attribute_val, flag)	17
IN	comm	communicator to which the attribute is attached (han-	18
		dle)	19
IN	comm_keyval	key value (integer)	20
	•		21
OUT	attribute_val	attribute value, unless $flag = false$	22
OUT	flag	false if no attribute is associated with the key (logical)	23
			24
int MPI_(25		
	int *flag)		26
MDT Comm	sot ottr(comm_comm_kours	al, attribute_val, flag, ierror) BIND(C)	27 ticket-248T.
	(MPI_Comm), INTENT(IN) ::		28
	GER, INTENT(IN) :: comm_		29
	-), INTENT(OUT) :: attribute_val	30 31
	CAL, INTENT(OUT) :: flag		32
	GER, OPTIONAL, INTENT(OUT		33
			34
	•	AL, ATTRIBUTE_VAL, FLAG, IERROR)	35
	GER COMM, COMM_KEYVAL, IE		36
	GER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	37
LUGI	CAL FLAG		38
{bool MP]	[::Comm::Get_attr(int com	m_keyval, void* attribute_val)	39
	const(binding deprecate	•	40
Dotai	was attribute value by lease	The coll is appropriate if there is no how with walve	41
		The call is erroneous if there is no key with value correct if the key value exists, but no attribute is	42
	,	case, the call returns $flag = false$. In particular	43
	AL_INVALID is an erroneous k		44
	ALL INVICED IS AIL CITOILCOUS K	oy varao.	45

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the location where the attribute value is to be returned. Thus, if the attribute value itself is 48

Unofficial Draft for Comment Only

	1 2 3 4	will	be of type void* and	hen the actual attribute_val parameter to MPI_Comm_set_attr d the actual attribute_val parameter to MPI_Comm_get_attr End of advice to users.)		
	5 6 7	void*	(*) avoids the messy	a formal parameter attribute_val or type void* (rather than type casting that would be needed if the attribute value is her than void*. (<i>End of rationale.</i>)		
	8 9 10 11		-	PI_ATTR_GET , whose use is deprecated. The C binding is g differs in that attribute_val is an address-sized integer.		
	12	MPI_COM	IM_DELETE_ATTR(comm, comm_keyval)		
	13 14 15	INOUT	comm	communicator from which the attribute is deleted (han- dle)		
	16 17	IN	comm_keyval	key value (integer)		
ticket-248T	18 19	int MPI_(Comm_delete_attr(MPI_Comm comm, int comm_keyval)		
	20	<pre>MPI_Comm_delete_attr(comm, comm_keyval, ierror) BIND(C)</pre>				
	21 22	TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: comm_keyval				
	23			TENT(OUT) :: ierror		
	24			, COMM_KEYVAL, IERROR)		
	25 26		GER COMM, COMM_KE			
	27 28	{void MP]	I::Comm::Delete_a Section 15.2)	<pre>ttr(int comm_keyval)(binding deprecated, see }</pre>		
	29	Delete	e attribute from cacl	ne by key. This function invokes the attribute delete function		
	30 31			when the keyval was created. The call will fail if the		
	32		_	returns an error code other than MPI_SUCCESS.		
	33			or is replicated using the function MPI_COMM_DUP, all call-		
	34			utes that are currently set are invoked (in arbitrary order).		
	35			deleted using the function MPI_COMM_FREE all callback that are currently set are invoked.		
	36			e as MPI_ATTR_DELETE but is needed to match the new		
	37			ns. The use of MPI_ATTR_DELETE is deprecated.		
	38	00111110				
	39 40	6.7.3 Wi	indows			
	40		· · · · · ·	· 1		
	42	1 ne new 1	functions for caching	on windows are:		
	43					
	44					
	45					
	46					
	47					
	48					

MPI_WI	N_CREATE_KEYVAL(win_c	opy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)	1
	· · · · ·		3
IN	win_copy_attr_fn	copy callback function for win_keyval (function)	4
IN	win_delete_attr_fn	delete callback function for win_keyval (function)	5
OUT	win_keyval	key value for future access (integer)	6
IN	extra_state	extra state for callback functions	7
			8
int MPI	_Win_create_keyval(MPI_	Win_copy_attr_function *win_copy_attr_fn,	9 10
	MPI_Win_delete_at	ttr_function *win_delete_attr_fn,	11
	<pre>int *win_keyval,</pre>	void *extra_state)	12 ticket-248T.
MPI_Win	_create_keyval(win_copy	<pre>v_attr_fn, win_delete_attr_fn, win_keyval,</pre>	13 UICKEU-2401.
	extra_state, ier	· · · · · · · · · · · · · · · · · · ·	14
PRO	CEDURE(MPI_Win_copy_att	r_function) :: win_copy_attr_fn	15
		<pre>attr_function) :: win_delete_attr_fn</pre>	16
	EGER, INTENT(OUT) :: w	•	17 18
		(IND), INTENT(IN) :: extra_state	19
	EGER, OPTIONAL, INTENT((UUI) :: lerror	20
MPI_WIN		_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	21
	EXTRA_STATE, IER		22
	ERNAL WIN_COPY_ATTR_FN,		23
	EGER WIN_KEYVAL, IERROF EGER(KIND=MPI_ADDRESS_K		24
			25
{static		<pre>xeyval(MPI::Win::Copy_attr_function*</pre>	26 27
	win_copy_attr_fn		28
		_attr_function* win_delete_attr_fn, e) (binding deprecated, see Section 15.2) }	29
			30
		n may be specified as MPI_WIN_NULL_COPY_FN or	31
		C++, or Fortran. MPI_WIN_NULL_COPY_FN is a	32
	8	han returning $flag = 0$ and MPI_SUCCESS. ded copy function that sets $flag = 1$, returns the value	33
	*	ut, and returns MPI_SUCCESS.	34 35
		fn may be specified as MPI_WIN_NULL_DELETE_FN	36
	-	MPI_WIN_NULL_DELETE_FN is a function that does	37
nothing,	other than returning MPI_S	SUCCESS.	38
The C ca	allback functions are:		39
twoodof	int MDT Win conv ottr	function (MDI Win oldwin int win kowyol	40
rypeder		_function(MPI_Win oldwin, int win_keyval, e, void *attribute_val_in,	41
		val_out, int *flag);	42
1			43 44
and typedef	int MDT Win doloto att	r_function(MPI_Win win, int win_keyval,	45
cypeder		val, void *extra_state);	46
		,	47
With the	e mpi_f08 module, the Fort	ran callback functions are:	48 ticket230-B.

ticket-248T.

```
1
           \mathbf{2}
                ABSTRACT INTERFACE
                  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
           3
                  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
           4
                       TYPE(MPI_Win) :: oldwin
           5
                       INTEGER :: win_keyval, ierror
           6
                       INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
           7
                       attribute_val_out
           8
                      LOGICAL :: flag
           9
           10
ticket230-B.
                and
           11
ticket-248T.
                ABSTRACT INTERFACE
           12
                  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
           13
                  extra_state, ierror) BIND(C)
           14
                       TYPE(MPI_Win) :: win
           15
                       INTEGER :: win_keyval, ierror
           16
                       INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
           17
           18
ticket230-B.
                The With the mpi module and mpif.h, the Fortran callback functions are:
           19
ticket250-V. 20
                SUBROUTINE WIN_COPY_ATTR_[FN]FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                               ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
           21
                    INTEGER OLDWIN, WIN_KEYVAL, IERROR
           22
                    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
           23
                         ATTRIBUTE_VAL_OUT
           24
                    LOGICAL FLAG
           25
           26
                and
           27
ticket250-V
                SUBROUTINE WIN_DELETE_ATTR_[FN] FUNCTION (WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
           28
                               EXTRA_STATE, IERROR)
           29
                    INTEGER WIN, WIN_KEYVAL, IERROR
           30
                     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
           ^{31}
           32
                The C++ callbacks are:
           33
                {typedef int MPI::Win::Copy_attr_function(const MPI::Win& oldwin,
           34
                               int win_keyval, void* extra_state, void* attribute_val_in,
           35
                               void* attribute_val_out, bool& flag); (binding deprecated, see
           36
                               Section 15.2
           37
           38
                and
           39
                {typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval,
           40
                               void* attribute_val, void* extra_state); (binding deprecated, see
           41
                               Section 15.2
           42
                    If an attribute copy function or attribute delete function returns other than
           43
                MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
           44
                erroneous.
           45
           46
           47
           48
```

MPI_WIN_FREE_KEYVAL(win_keyval)					
INOUT	win_keyval	key value (integer)	2 3		
			4		
int MPI_Wi	n_free_keyval(int *win_k	eyval)	5 ticket-248T.		
MPI_Win_free_keyval(win_keyval, ierror) BIND(C)					
	R, INTENT(INOUT) :: win	•	7 8		
INTEGE	R, OPTIONAL, INTENT(OUT)	:: ierror	9		
MPI_WIN_FF INTEGE	10 11				
{static vo	oid MPI::Win::Free_keyval	(int& win_keyval) (binding deprecated, see	12		
-	Section 15.2 }		13		
			14 15		
			16		
	SET_ATTR(win, win_keyval, at	,	17		
INOUT	win	window to which attribute will be attached (handle)	18		
IN	win_keyval	key value (integer)	19		
IN	attribute_val	attribute value	20 21		
			22		
int MPI_Wi	n_set_attr(MPI_Win win,	<pre>int win_keyval, void *attribute_val)</pre>	23 ticket-248T.		
MPI_Win_se	et_attr(win, win_keyval,	attribute_val, ierror) BIND(C)	24		
	<pre>IPI_Win), INTENT(IN) ::</pre>		25 26		
	R, INTENT(IN) :: win_ke	•	27		
	R(KIND=MPI_ADDRESS_KIND) R, OPTIONAL, INTENT(OUT)	, INTENT(IN) :: attribute_val	28		
			29		
	T_ATTR(WIN, WIN_KEYVAL, R WIN, WIN_KEYVAL, IERRO		30		
	R(KIND=MPI_ADDRESS_KIND)		31 32		
		_	33		
{void MPI:	<pre>:win::Set_attr(int win_k</pre>	eyval, const void* attribute_val)(binding	34		
	acprecatea, see section 1		35		
			36		
MPI_WIN_	GET_ATTR(win, win_keyval, a	ttribute_val, flag)	37 38		
IN	win	window to which the attribute is attached (handle)	39		
IN	win_keyval	key value (integer)	40		
OUT	attribute_val	attribute value, unless $flag = false$	41		
			42		
OUT	flag	false if no attribute is associated with the key (logical)	43 44		
int MPT Wi	n get attr(MPT Win win	int win_keyval, void *attribute_val,	45		
+_ // 1	int *flag)	,,,	46		
MPT Win or	J.	attribute_val, flag, ierror) BIND(C)	$_{47}$ ticket-248T.		
THE T _ M TH _ Be	Julator (WIII, WIII_Keyval,	accitouce_vat, itag, terror) DIND(C)	48		

```
1
                     TYPE(MPI_Win), INTENT(IN) :: win
            \mathbf{2}
                     INTEGER, INTENT(IN) :: win_keyval
            3
                     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
            4
                     LOGICAL, INTENT(OUT) :: flag
            5
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            6
                MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
            7
                     INTEGER WIN, WIN_KEYVAL, IERROR
            8
                     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
            9
                     LOGICAL FLAG
           10
           11
                 {bool MPI::Win::Get_attr(int win_keyval, void* attribute_val) const(binding
           12
                                deprecated, see Section 15.2 }
           13
           14
           15
                 MPI_WIN_DELETE_ATTR(win, win_keyval)
           16
                   INOUT
                                                        window from which the attribute is deleted (handle)
                            win
           17
           18
                   IN
                            win_keyval
                                                        key value (integer)
           19
           20
                 int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
ticket-248T. 21
                 MPI_Win_delete_attr(win, win_keyval, ierror) BIND(C)
           22
                     TYPE(MPI_Win), INTENT(IN) :: win
           23
                     INTEGER, INTENT(IN) :: win_keyval
           24
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           25
           26
                 MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
           27
                     INTEGER WIN, WIN_KEYVAL, IERROR
           28
                 {void MPI::Win::Delete_attr(int win_keyval)(binding deprecated, see Section 15.2)
           29
                                }
           30
           ^{31}
           32
                 6.7.4 Datatypes
           33
           34
                 The new functions for caching on datatypes are:
           35
           36
                 MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)
           37
           38
           39
                   IN
                            type_copy_attr_fn
                                                        copy callback function for type_keyval (function)
           40
                   IN
                            type_delete_attr_fn
                                                        delete callback function for type_keyval (function)
           41
                   OUT
                            type_keyval
                                                        key value for future access (integer)
           42
           43
                   IN
                            extra_state
                                                        extra state for callback functions
           44
           45
                 int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
           46
                                MPI_Type_delete_attr_function *type_delete_attr_fn,
           47
                                int *type_keyval, void *extra_state)
ticket-248T. 48
```

```
1
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
                                                                                       2
              extra_state, ierror) BIND(C)
                                                                                       3
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
                                                                                       4
    INTEGER, INTENT(OUT) :: type_keyval
                                                                                       5
                                                                                       6
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       7
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
                                                                                       9
              EXTRA_STATE, IERROR)
                                                                                       10
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
                                                                                       11
    INTEGER TYPE_KEYVAL, IERROR
                                                                                       12
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                       13
                                                                                       14
{static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
                                                                                       15
              type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
                                                                                       16
              type_delete_attr_fn, void* extra_state) (binding deprecated, see
                                                                                       17
              Section 15.2 }
                                                                                      18
    The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or
                                                                                      19
MPI_TYPE_DUP_FN from either C, C++, or Fortran. MPI_TYPE_NULL_COPY_FN is a
                                                                                      20
function that does nothing other than returning flag = 0 and MPI_SUCCESS.
                                                                                      21
MPI_TYPE_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value
                                                                                      22
of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.
                                                                                      23
    The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN
                                                                                       24
from either C, C++, or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does
                                                                                       25
nothing, other than returning MPI_SUCCESS.
                                                                                       26
The C callback functions are:
                                                                                       27
                                                                                       28
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
                                                                                      29
              int type_keyval, void *extra_state, void *attribute_val_in,
                                                                                       30
              void *attribute_val_out, int *flag);
                                                                                       31
                                                                                      32
and
                                                                                      ^{33} ticket252-W.
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
                                                                                      34
              int type_keyval, void *attribute_val, void *extra_state);
                                                                                      35
With the mpi_f08 module, the Fortran callback functions are:
                                                                                      <sup>36</sup> ticket230-B.
                                                                                      <sup>37</sup> ticket-248T.
ABSTRACT INTERFACE
                                                                                      38
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                      39
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                       40
      TYPE(MPI_Datatype) :: oldtype
                                                                                       41
      INTEGER :: type_keyval, ierror
                                                                                       42
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                       43
      attribute_val_out
                                                                                       44
      LOGICAL :: flag
                                                                                       45
                                                                                      <sup>46</sup> ticket230-B.
and
ABSTRACT INTERFACE
                                                                                      <sup>47</sup> ticket-248T.
                                                                                       48
```

```
1
                   SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
           \mathbf{2}
                   attribute_val, extra_state, ierror) BIND(C)
            3
                       TYPE(MPI_Datatype) :: datatype
            4
                       INTEGER :: type_keyval, ierror
           5
                       INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
            6
ticket230-B. 7
                 The With the mpi module and mpif.h, the Fortran callback functions are:
ticket250-V.
                 SUBROUTINE TYPE_COPY_ATTR_[FN] FUNCTION (OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                               ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
           10
                     INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
           11
                     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
           12
                         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
           13
                     LOGICAL FLAG
           14
           15
                 and
                SUBROUTINE TYPE_DELETE_ATTR_[FN] FUNCTION (DATATYPE, TYPE_KEYVAL,
ticket250-V.<sup>16</sup>
ticket252-W. 17
                               ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
ticket252-W. 18
                     INTEGER DATATYPE, TYPE_KEYVAL, IERROR
           19
                     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
           20
                The C++ callbacks are:
           21
           22
                 {typedef int
           23
                               MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype,
           24
                               int type_keyval, void* extra_state,
           25
                               const void* attribute_val_in, void* attribute_val_out,
           26
                               bool& flag); (binding deprecated, see Section 15.2)}
           27
                 and
           28
ticket252-W. 29
                 {typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& datatype,
                               int type_keyval, void* attribute_val, void* extra_state);
           30
                               (binding deprecated, see Section 15.2)}
           ^{31}
           32
                     If an attribute copy function or attribute delete function returns other than
           33
                 MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
           34
                 is erroneous.
           35
           36
           37
                 MPI_TYPE_FREE_KEYVAL(type_keyval)
           38
                            type_keyval
                  INOUT
                                                       key value (integer)
           39
           40
                 int MPI_Type_free_keyval(int *type_keyval)
           41
ticket-248T.
           42
                 MPI_Type_free_keyval(type_keyval, ierror) BIND(C)
           43
                     INTEGER, INTENT(INOUT) :: type_keyval
           44
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           45
                MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
           46
                     INTEGER TYPE_KEYVAL, IERROR
           47
           48
```

{static v	<pre>void MPI::Datatype::Free see Section 15.2) }</pre>	_keyval(int& type_keyval)(binding deprecated,	1 2 3
INOUT	E_SET_ATTR(<mark>data</mark> type, type [ticket252-W.] <mark>data</mark> type	datatype to which attribute will be attached (handle)	⁴ ⁵ ticket252-W. ⁷
IN IN	type_keyval attribute_val	key value (integer) attribute value	8 9 10
int MPI_7	Type_set_attr(MPI_Dataty void *attribute_val	pe <mark>data</mark> type, int type_keyval, .)	$^{11}_{12}$ ticket252-W.
TYPE (INTEC INTEC	(MPI_Datatype), INTENT(I GER, INTENT(IN) :: type	_keyval D), INTENT(IN) :: attribute_val	14 15 16 17 18
INTEC	_SET_ATTR(<mark>DATA</mark> TYPE, TYPE GER <mark>DATA</mark> TYPE, TYPE_KEYVA GER(KIND=MPI_ADDRESS_KIN	•	$^{19}_{20}$ ticket252-W. $^{21}_{21}$ ticket252-W.
{void MP]	• =	t type_keyval, const void* ing deprecated, see Section 15.2) }	23 24 25 26
MPI_TYPI	E_GET_ATTR(<mark>data</mark> type, type	e_keyval, attribute_val, flag)	27 ticket252-W.
IN	[ticket252-W.] <mark>data</mark> type	datatype to which the attribute is attached (handle)	28 29
IN	type_keyval	key value (integer)	30
OUT	attribute_val	attribute value, unless $flag = false$	31
OUT	flag	false if no attribute is associated with the key (logical)	32 33 34
int MPI_7	<pre>Sype_get_attr(MPI_Dataty *attribute_val, int</pre>	pe <mark>data</mark> type, int type_keyval, void ; *flag)	³⁵ ticket252-W. ³⁶ ₃₇ ticket-248T.
• •	BIND(C)	_keyval, attribute_val, flag, ierror)	37 UCKCU-2401. 38 39
INTEC INTEC LOGIC	(MPI_Datatype), INTENT(I GER, INTENT(IN) :: type GER(KIND=MPI_ADDRESS_KIN CAL, INTENT(OUT) :: fla GER, OPTIONAL, INTENT(OU	_keyval D), INTENT(OUT) :: attribute_val g	40 41 42 43 44
INTEC INTEC	_GET_ATTR(DATATYPE, TYPE GER DATATYPE, TYPE_KEYVA GER(KIND=MPI_ADDRESS_KIN CAL FLAG	-	 ⁴⁵ ticket252-W. ⁴⁶ ticket252-W. ⁴⁷ ⁴⁸

```
292
                          CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING
            1
                 {bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val)
            \mathbf{2}
                                const(binding deprecated, see Section 15.2)
            3
            4
            5
ticket252-W.
                 MPI_TYPE_DELETE_ATTR(datatype, type_keyval)
            6
                   INOUT
                            [ticket252-W.]datatype
                                                        datatype from which the attribute is deleted (handle)
            \overline{7}
                            type_keyval
            8
                   IN
                                                        key value (integer)
            9
ticket252-W. ^{10}
                 int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
ticket-248T.<sup>11</sup>
                 MPI_Type_delete_attr(datatype, type_keyval, ierror) BIND(C)
            12
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
            13
                     INTEGER, INTENT(IN) :: type_keyval
           14
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            15
ticket252-W. ^{16}
                 MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
ticket252-W. ^{17}
                     INTEGER DATATYPE, TYPE_KEYVAL, IERROR
            18
                 {void MPI::Datatype::Delete_attr(int type_keyval)(binding deprecated, see
           19
                                Section 15.2 }
           20
           21
           22
                        Error Class for Invalid Keyval
                 6.7.5
           23
           24
                 Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL.
           25
                 Only such values can be passed to the functions that use key values as input arguments.
           26
                 In order to signal that an erroneous key value has been passed to one of these functions,
                 there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by
           27
                 MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE,
           28
                 MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR,
           29
                 MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL,
           30
                 MPI_COMM_DUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last three are
           31
                 included because keyval is an argument to the copy and delete functions for attributes.
           32
           33
           34
                 6.7.6 Attributes Example
           35
                                          This example shows how to write a collective communication
                      Advice to users.
           36
                      operation that uses caching to be more efficient after the first call. The coding style
           37
                      assumes that MPI function results return only error statuses. (End of advice to users.)
           38
           39
                    /* key for this module's stuff: */
            40
                    static int gop_key = MPI_KEYVAL_INVALID;
           41
           42
                    typedef struct
           43
                    {
           44
                        int ref_count;
                                                  /* reference count */
            45
                        /* other stuff, whatever else we want */
            46
                    } gop_stuff_type;
            47
            48
```

```
1
void Efficient_Collective_Op (MPI_Comm comm, ...)
                                                                                 2
{
                                                                                 3
  gop_stuff_type *gop_stuff;
  MPI_Group
                                                                                 4
                   group;
                                                                                 5
  int
                   foundflag;
                                                                                 6
                                                                                 7
  MPI_Comm_group(comm, &group);
  if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
                                                                                 9
                                                                                 10
  {
                                                                                 11
    if ( ! MPI_Comm_create_keyval( gop_stuff_copier,
                              gop_stuff_destructor,
                                                                                 12
                               &gop_key, (void *)0));
                                                                                 13
    /* get the key while assigning its copy and delete callback
                                                                                 14
                                                                                 15
       behavior. */
                                                                                 16
                                                                                 17
    MPI_Abort (comm, 99);
                                                                                 18
  }
                                                                                 19
                                                                                 20
  MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
                                                                                 21
  if (foundflag)
  { /* This module has executed in this group before.
                                                                                 22
                                                                                 23
       We will use the cached information */
  }
                                                                                 24
                                                                                 25
  else
                                                                                 26
  { /* This is a group that we have not yet cached anything in.
       We will now do so.
                                                                                 27
    */
                                                                                 28
                                                                                 29
                                                                                 30
    /* First, allocate storage for the stuff we want,
       and initialize the reference count */
                                                                                 31
                                                                                 32
                                                                                 33
    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
                                                                                 34
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
                                                                                 35
    gop_stuff -> ref_count = 1;
                                                                                 36
                                                                                 37
                                                                                 38
    /* Second, fill in *gop_stuff with whatever we want.
                                                                                 39
       This part isn't shown here */
                                                                                 40
                                                                                 41
    /* Third, store gop_stuff as the attribute value */
                                                                                 42
    MPI_Comm_set_attr ( comm, gop_key, gop_stuff);
  }
                                                                                 43
                                                                                 44
  /* Then, in any case, use contents of *gop_stuff
                                                                                 45
     to do the global op ... */
                                                                                 46
}
                                                                                 47
                                                                                 48
/* The following routine is called by MPI when a group is freed */
```

```
1
           \mathbf{2}
                    int gop_stuff_destructor (MPI_Comm comm, int keyval, void *gop_stuffP,
            3
                                            void *extra)
                    {
           4
           5
                      gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
            6
                      if (keyval != gop_key) { /* abort -- programming error */ }
            7
            8
                      /* The group's being freed removes one reference to gop_stuff */
           9
                      gop_stuff -> ref_count -= 1;
           10
           11
                      /* If no references remain, then free the storage */
           12
                      if (gop_stuff -> ref_count == 0) {
           13
                        free((void *)gop_stuff);
           14
                      }
           15
                      return MPI_SUCCESS;
           16
                    }
           17
           18
                    /* The following routine is called by MPI when a group is copied */
           19
                    int gop_stuff_copier (MPI_Comm comm, int keyval, void *extra,
           20
                     void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
           21
                    {
           22
                      gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
           23
                      gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
           24
                      if (keyval != gop_key) { /* abort -- programming error */ }
           25
           26
                      /* The new group adds one reference to this gop_stuff */
           27
                      gop_stuff_in -> ref_count += 1;
           28
                      *gop_stuff_out = gop_stuff_in;
           29
                      return MPI_SUCCESS;
           30
                    }
           ^{31}
           32
                       Naming Objects
                 6.8
           33
           34
                 There are many occasions on which it would be useful to allow a user to associate a printable
           35
                identifier with an MPI communicator, window, or datatype, for instance error reporting,
           36
                 debugging, and profiling. The names attached to opaque objects do not propagate when
           37
                 the object is duplicated or copied by MPI routines. For communicators this can be achieved
           38
                using the following two functions.
           39
           40
           41
                 MPI_COMM_SET_NAME (comm, comm_name)
           42
                  INOUT
                            comm
                                                       communicator whose identifier is to be set (handle)
           43
                  IN
                                                       the character string which is remembered as the name
                            comm_name
           44
                                                       (string)
           45
           46
  ticket140. 47
                int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
ticket-248T. 48
```

	_set_name(comm, comm_			
	<pre>(MPI_Comm), INTENT(IN ACTER(LEN=*), INTENT(</pre>			
	GER, OPTIONAL, INTENT			
MPI_COMM	_SET_NAME(COMM, COMM_	NAME, IERROR)		
	GER COMM, IERROR	6 7		
CHAR.	ACTER*(*) COMM_NAME	8		
{void MP	I::Comm::Set_name(con Section 15.2) }	st char* comm_name) (binding deprecated, see 9		
The chara MPI libra stack). Le MPI_ name of th call. Ther	acter string which is pass ry (so it can be freed by reading spaces in name are COMM_SET_NAME is a he communicator as seen in	vs a user to associate a name string with a communicator. ed to MPI_COMM_SET_NAME will be saved inside the the caller immediately after the call, or allocated on the significant but trailing ones are not. local (non-collective) operation, which only affects the in the process which made the MPI_COMM_SET_NAME the same (or any) name be assigned to a communicator		
is se		COMM_SET_NAME is provided to help debug code, it ame to a communicator in all of the processes where it and of advice to users.)		
MPI_MAX_ low for the	_OBJECT_NAME in Fortra e null terminator. Attemp	h can be stored is limited to the value of 24 an and MPI_MAX_OBJECT_NAME-1 in C and C++ to al- bits to put names longer than this will result in truncation 26 AME must have a value of at least 64. 27		
of a view	ny length could fail, then ved only as a strict upper	amstances of store exhaustion an attempt to put a name refore the value of MPI_MAX_OBJECT_NAME should be bound on the name length, not a guarantee that setting a will always succeed. (<i>End of advice to users.</i>)		
Advice to implementors. Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. Implementations which allocate space for the name from the heap should still define MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate space for a string of up to this size when calling MPI_COMM_GET_NAME. (End of advice to implementors.)				
MPI_COM	1M_GET_NAME (comm,	comm_name, resultlen) 40		
IN	comm	communicator whose name is to be returned (handle) $\frac{42}{43}$		
OUT	comm_name	the name previously stored on the communicator, or an empty string if no such name exists (string) 43		
OUT	resultlen	length of returned name (integer) 46		
int MPI_	Comm_get_name(MPI_Com	m comm, char *comm_name, int *resultlen) 47		

	1	
ticket-248T.		<pre>MPI_Comm_get_name(comm, comm_name, resultlen, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm</pre>
	4	CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
	5 6	INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	7	
	8 9	MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR) INTEGER COMM, RESULTLEN, IERROR
	10	CHARACTER*(*) COMM_NAME
	11 12	<pre>{void MPI::Comm::Get_name(char* comm_name, int& resultlen) const(binding</pre>
	13	MPI_COMM_GET_NAME returns the last name which has previously been associated
	14	with the given communicator. The name may be set and got from any language. The same
	15 16	name will be returned independent of the language used. name should be allocated so that
	17	it can hold a resulting string of length $MPI_MAX_OBJECT_NAME$ characters.
	18	MPI_COMM_GET_NAME returns a copy of the set name in name.
	19	In C, a null character is additionally stored at name[resultlen]. The value of resultlen
ticket207. ticket207.	20	cannot be larger [then]than MPI_MAX_OBJECT_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger [then]than
ticket207.	21	MPI_MAX_OBJECT_NAME.
	22	If the user has not associated a name with a communicator, or an error occurs,
	23 24	MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C and
	24 25	C++). The three predefined communicators will have predefined names associated with
	26	them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator
	27	returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of
	28	MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from
	29	setting a name on the same communicator; doing this removes the old name and assigns
	30	the new one.
	31 32	
	33	Rationale. We provide separate functions for setting and getting the name of a com-
	34	municator, rather than simply providing a predefined attribute key for the following
	35	reasons:
	36	• It is not, in general, possible to store a string as an attribute from Fortran.
	37	• It is not easy to set up the delete function for a string attribute unless it is known
	38	to have been allocated from the heap.
	39 40	• To make the attribute key useful additional code to call strdup is necessary. If
	40	this is not standardized then users have to write it. This is extra unneeded work
	42	which we can easily eliminate.
	43	• The Fortran binding is not trivial to write (it will depend on details of the
	44	Fortran compilation system), and will not be portable. Therefore it should be in
	45	the library rather than in user code.
	46	(End of rationale.)
	47 48	

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (End of advice to users.)

 12 ticket 219. The following functions are used for setting and getting names of datatypes. The constant MPI_MAX_OBJECT_NAME also applies to these names. 14

			10
MPI_TYPI	E_SET_NAME (<mark>data</mark> type, type	_name)	16 ticket252-W.
INOUT	[ticket252-W.] <mark>data</mark> type	datatype whose identifier is to be set (handle)	17
		· -	18
IN	type_name	the character string which is remembered as the name	19
		(string)	20
			21
int MPI_7	<pre>Sype_set_name(MPI_Datatype</pre>	e <mark>data</mark> type, <mark>const</mark> char *type_name)	$_{22}$ ticket252-W.
MDT Turne	_set_name(datatype, type_1	name ierror) RIND(C)	$_{23}$ ticket 140.
			$_{24}$ ticket-248T.
	(MPI_Datatype), INTENT(IN)	• -	25
	ACTER(LEN=*), INTENT(IN)		26
INTEC	SER, OPTIONAL, INTENT(OUT)) :: lerror	27
MPI_TYPE_	_SET_NAME(DATATYPE, TYPE_I	NAME, IERROR)	²⁸ ticket252-W.
INTEG	ER DATATYPE, IERROR		²⁹ ticket252-W.
CHARA	ACTER*(*) TYPE_NAME		30
6			31
{void MP1	• -	<pre>st char* type_name)(binding deprecated, see</pre>	32
	Section 15.2 }		33
			34
			35
MPI_TYPI	E_GET_NAME (<mark>data</mark> type, type	_name, resultlen)	$_{36}$ ticket252-W.
IN	[ticket252-W.] <mark>data</mark> type	datatype whose name is to be returned (handle)	37
OUT			38
001	type_name	the name previously stored on the datatype, or a empty	39
		string if no such name exists (string)	40
OUT	resultlen	length of returned name (integer)	41
			42
int MPI_7	Type_get_name(MPI_Datatype	e <mark>data</mark> type, char *type_name, int	43 ticket252-W.
	*resultlen)		44
			$_{45}$ ticket-248T.
		name, resultlen, ierror) BIND(C)	46
	(MPI_Datatype), INTENT(IN)	• -	47
CHARA	ACTER(LEN=MPI_MAX_OBJECT_I	NAME), INTENT(OUT) :: type_name	48

1 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

13

	1 2		ER, INTENT(OUT) :: res ER, OPTIONAL, INTENT(OU	
ticket252-W. ticket252-W.		INTEG	GET_NAME(DATATYPE, TYPE ER DATATYPE, RESULTLEN, CTER*(*) TYPE_NAME	_NAME, RESULTLEN, IERROR) IERROR
	7 8	{void MPI	::Datatype::Get_name(cha deprecated, see Section	ar* type_name, int& resultlen) const(binding 15.2) }
ticket219.		ple, MPI_V The fo	VCHAR has the default name	or setting and getting names of windows. The con-
	15	MPI_WIN_	SET_NAME (win, win_name)
	16 17	INOUT	win	window whose identifier is to be set (handle)
	18 19 20	IN	win_name	the character string which is remembered as the name (string)
ticket140.	21	int MPI_W	in_set_name(MPI_Win win	, const char *win_name)
ticket-248T.	22 23 24 25 26	TYPE (CHARA	et_name(win, win_name, : MPI_Win), INTENT(IN) :: CTER(LEN=*), INTENT(IN) ER, OPTIONAL, INTENT(OU	win :: win_name
	27 28 29 30	INTEG	ET_NAME(WIN, WIN_NAME, 1 ER WIN, IERROR CTER*(*) WIN_NAME	IERROR)
	31 32 33	{void MPI	::Win::Set_name(const cl Section 15.2) }	<pre>nar* win_name)(binding deprecated, see</pre>
	34 35	MPI WIN		e resultien)
	36	IN	win	window whose name is to be returned (handle)
	37 38 39	OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)
	40 41	OUT	resultlen	length of returned name (integer)
1.1.4.9.40 T	42	int MPI_W	in_get_name(MPI_Win win	, char *win_name, int *resultlen)
ticket-248T.		TYPE (CHARA	<pre>MPI_Win), INTENT(IN) ::</pre>	_NAME), INTENT(OUT) :: win_name
	48		ER, OPTIONAL, INTENT(OU	

MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
INTEGER WIN, RESULTLEN, IERROR
CHARACTER*(*) WIN_NAME

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communicators will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static communicator allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can

Unofficial Draft for Comment Only

 31

¹ be concurrently active in each processor, then it is sufficient to allocate one communicator
 ² per library.

3 4

5

19

20

21 22

23

24 25

26

28

34

Dynamic communicator allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in
 a subset of a group executing the same parallel procedure. Thus, processes that execute
 the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

• messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;

- messages are always selected by source (no use is made of MPI_ANY_SOURCE).

ticket0. ²⁷ The General [c]Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, then communicator creation be properly coordinated.

- 35

 36

 37

 38

 39

 40

 41

 42

 43

 44

 45

 46

 47
- 48

Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to intercommunicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal [only]with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [44]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [11, 12].

 24

 31

ticket258.

ticket258

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem [then]than that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord (0,0): rank 0 coord (0,1): rank 1 coord (1,0): rank 2 coord (1,1): rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

ticket259. 42

7.4 Overview of the Functions

⁴³ [The functions MPI_GRAPH_CREATE, MPI_DIST_GRAPH_CREATE_ADJACENT,
⁴⁴ MPI_DIST_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph)
⁴⁶ virtual topologies and Cartesian topologies, respectively. These topology creation functions
⁴⁷ are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.]MPI supports three topology types: Cartesian,

Unofficial Draft for Comment Only

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

24

25

1

2

- ticket0. $\frac{22}{23}$
 - 26 27

28 29

30 31 32

33 34

35 36

37

38

39

graph, and distributed graph. The function MPI_CART_CREATE is used to create Cartesian topologies, the function MPI_GRAPH_CREATE is used to create graph topologies, and the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to create distributed graph topologies. 4

5The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For 6 $\overline{7}$ MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. [For MPI_DIST_GRAPH_CREATE_ADJACENT ⁸ ticket259 9 and MPI_DIST_GRAPH_CREATE the input communication graph is distributed across the 10 calling processes. When calling MPI_GRAPH_CREATE, each process specifies all nodes and 11edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or 12MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby 13 each process only specifies a subset of the edges in the graph such that the entire graph 14structure is defined collectively across the set of processes. Therefore the processes pro-15vide different values for the arguments specifying the graph. However, all processes must 16give the same value for reorder and the info argument. In all cases, a new communica-17 tor comm_topol is created that carries the topological structure as cached information (see 18 Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propa-19 gates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an *n*-dimensional hypercube is an *n*-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

Rationale. Similar functions are contained in EXPRESS [13] and PARMACS. (*End of rationale.*)

The function MPI_TOPO_TEST can be used to inquire about the topology associated 30 with a communicator. The topological information can be extracted from the communicator 31using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and 32 MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional 33 functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK 34 and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-35 versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analo-36 gous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information 37 needed to communicate with neighbors in a Cartesian dimension. The two functions 38 MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract 39 the neighbors of a node in a graph. For distributed graphs, the functions 40 MPI_DIST_NEIGHBORS_COUNT and MPI_DIST_NEIGHBORS can be used to extract the 41 neighbors of the calling node. The function MPI_CART_SUB is collective over the in-42put communicator's group; all other functions are local.] MPI defines functions to query 43 a communicator for topology information. The function MPI_TOPO_TEST is used to 44query for the type of topology associated with a communicator. Depending on the topol-45ogy type, different information can be extracted. For a graph topology, the functions 46 MPI_GRAPHDIMS_GET and MPI_GRAPH_GET return the values that were specified in the 47call to MPI_GRAPH_CREATE. Additionally, the functions MPI_GRAPH_NEIGHBORS_COUNT 48

303

```
^{29} ticket 259.
```

20

21

22

23 24

25

26

27

and MPI_GRAPH_NEIGHBORS can be used to obtain the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CART_COREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_CORESTE. Additionally, the functions MPI_CART_RANK and MPI_CART_CORESTE analate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Carte- sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an imple- mentation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLCOALL, MPI_NEIGHBOR_ALLCOALLV, and MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLCOALL, MPI_NEIGHBOR_ALLCOALLV, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLCOALLV, and MPI_INEIGHBOR_ALLCOALLW communicate with the nearest neighbors on the topol- ogy associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_INEIGHBOR_ALLTOALLW. MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) NPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm
 MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions. Section 7.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLCALLWV, and MPI_NEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLCALLWV, and MPI_INEIGHBOR_ALLCALLWV communicator. The nonblocking variants are MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_ET return the values that were specified in the call to MPI_CART_CCEATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Carte- sian subspace (analogous to MPI_CORM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an imple- mentation. The neighborhood collective communicate with the nearest neighbors on the topol- ogy associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol- ogy associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Carte- sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an imple- mentation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCATHEV, MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCALLEV, and MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Carte- sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an imple- mentation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLCATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLCATHERV, and MPI_NEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLW. T.5.1 Cartesian Constructors T.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Carte- sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an imple- mentation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCALL, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLCALL, MPI_INEIGHBOR_ALLCOALLV, and MPI_INEIGHBOR_ALLCALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLV, MPI_INEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALIV, and MPI_INE
 ticket259. ineitherion in a Cartesian dimension. All of these query functions are local. For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. The neighborALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLCATHERV, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLCATHER, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLCATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
Image: Proceeding of the second system of
 sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALLW, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALLW, and MPI_NEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 communicator's group. Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an imple- mentation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol- ogy associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
13 Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in 14 the last section. In general these functions are not called by the user directly. However, 15 together with the communicator manipulation functions presented in Chapter 6, they are 16 sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. 18 The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, 19 MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, 20 and MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, 21 ogy associated with the communicator. The nonblocking variants are 22 MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 23 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 24 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 25 7.5 Topology Constructors 26 7.5.1 Cartesian Constructor 31 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) 32 IN comm_old 33 IN comm_old
 the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALLW, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALLW, and MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
ticket258. ¹⁷ mentation. ¹⁸ The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, ¹⁹ MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, ²⁰ and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol- ²¹ ogy associated with the communicator. The nonblocking variants are ²² MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, ²³ MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLGATHERV, ²⁴ MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and ²⁵ ²⁶ 7.5 Topology Constructors ²⁷ 7.5.1 Cartesian Constructor ³⁰ ³¹ MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) ³³ IN comm_old input communicator (handle)
18 The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, 19 MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, 20 and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol- 21 ogy associated with the communicator. The nonblocking variants are 22 MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 23 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 24 MPI_INEIGHBOR_ALLTOALLW. 25 7.5 26 7.5 31 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) 33 IN comm_old 34 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
 MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol- ogy associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol- ogy associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
21 ogy associated with the communicator. The nonblocking variants are 22 MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 23 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 24 MPI_INEIGHBOR_ALLTOALLW. 25 7.5 26 7.5 27 7.5 28 7.5.1 29 7.5.1 31 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) 32 IN 33 IN 44 input communicator (handle)
 MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 MPI_INEIGHBOR_ALLTOALLW. 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 7.5 Topology Constructors 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
7.5 Topology Constructors 7.5 Topology Constructor 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) N comm_old input communicator (handle)
 7.5.1 Cartesian Constructor MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
7.5.1 Cartesian Constructor ²⁹ ³⁰ ³¹ ³² MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) ³³ IN comm_old input communicator (handle)
 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
 MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart) IN comm_old input communicator (handle)
³⁴ IN ndims number of dimensions of Cartesian grid (integer)
³⁵ IN dims integer array of size ndims specifying the number of
³⁶ and ³⁷ processes in each dimension
³⁸ IN periods logical array of size ndims specifying whether the grid
is periodic (true) or not (false) in each dimension
⁴⁰ IN reorder ranking may be reordered (true) or not (false) (logical)
⁴⁰ IN reorder ranking may be reordered (true) or not (false) (logical)
41
⁴¹ OUT comm_cart communicator with new Cartesian topology (handle)
⁴¹ OUT comm_cart communicator with new Cartesian topology (handle) ⁴³ ticket140. int MPI_Cart_create(MPI_Comm_comm_old, int_ndims, const_int [*dims]dims[],
41 OUT comm_cart communicator with new Cartesian topology (handle) 42 ticket140. int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int [*dims]dims[], ticket126. const int [*periods]periods[], int reorder,
41 42OUT 43 ticket140. 44 ticket126. 45 ticket126.OUT 45 46comm_cartcommunicator with new Cartesian topology (handle)41 42OUT 43 ticket126. 45 ticket126.out 44 46comm_cartcommunicator with new Cartesian topology (handle)41 42int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int [*dims]dims[], const int [*periods]periods[], int reorder, MPI_Comm *comm_cart)
41 OUT comm_cart communicator with new Cartesian topology (handle) 42 ticket140. int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int [*dims]dims[], ticket126. const int [*periods]periods[], int reorder, ticket140. MPI_Comm *comm_cart)

TYPE(MPI_Comm), INTENT(IN) :: comm_old INTEGER, INTENT(IN) :: ndims, dims(ndims) LOGICAL, INTENT(IN) :: periods(ndims), reorder TYPE(MPI_Comm), INTENT(OUT) :: comm_cart INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2 3 4 5
<pre>MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR) INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR LOGICAL PERIODS(*), REORDER</pre>	6 7 8 9
<pre>{MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],</pre>	10 11 12
MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of [comm_comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.	13 14 15 16 17 18 19 ticket0. 20 21 22

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an *n*-dimensional topology.

MPI_DIMS	MPI_DIMS_CREATE(nnodes, ndims, dims)		
IN	nnodes	number of nodes in a grid (integer)	33
			34
IN	ndims	number of Cartesian dimensions (integer)	35
INOUT	dims	integer array of size ndims specifying the number of	36
		nodes in each dimension	37
			38
int MPI D:	ims_create(int nnodes, i	nt ndims. int *dims)	39
_			40 ticket-248T.
MPI_Dims_	create(nnodes, ndims, di	ms, ierror) BIND(C)	41
INTEG	ER, INTENT(IN) :: nnode	s, ndims	42
	ER, INTENT(INOUT) :: di		43
INTEG	ER, OPTIONAL, INTENT(OUT	44	
MPT DIMS (S_CREATE(NNODES, NDIMS, DIMS, IERROR)		45
	ER NNODES, NDIMS, DIMS(*		46
1.110		,, <u></u>	47
	48		

MPL DIMS_CREATE(nnodes_ndims_dims)

23

 24 25

26

27

 28

29

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i; only those entries where dims[i] = 0 are modified by the call.

9 10 11

12

13

14

15 16

27 28 29

30

1

 $\mathbf{2}$

3

4

5

6

7

8

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a multiple of $\prod dims[i]$.

 $i,dims[i] \neq 0$

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.

Example 7.1

dims	function call	dims
before call		on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

ticket259. ²⁶

7.5.3 [General (Graph)]Graph Constructor

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

	31	IN	comm_old	input communicator (handle)
	32	IN	nnodes	number of nodes in graph (integer)
	33	IN	index	array of integers describing node degrees (see below)
	34 35	IN	edges	array of integers describing graph edges (see below)
	36	IN	reorder	ranking may be reordered (true) or not (false) (logical)
	37	OUT	comm_graph	communicator with graph topology added (handle)
	38 39		-0 1	
ticket140.	• 40	int MPI_G	raph_create(MPI_Comm comm	n_old, int nnodes, <mark>const</mark>
ticket125.	• 41		int [*index]index[],	<pre>const int [*edges]edges[], int reorder,</pre>
ticket140.	• 42		MPI_Comm *comm_graph)
ticket125.		MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,		
ticket-248T.	• 43 • 44	MP1_Graph_		, index, edges, reorder, comm_graph,
			ierror) BIND(C)	
	45	TYPE(N	<pre>MPI_Comm), INTENT(IN) ::</pre>	comm_old
	46	INTEGH	ER, INTENT(IN) :: nnode:	s, index(nnodes), edges(*)
	47	TOCTC	T TNTENT(TN) roord	

47 LOGICAL, INTENT(IN) :: reorder 48 TYPE(MPL Comm) INTENT(OUT) ...

TYPE(MPI_Comm), INTENT(OUT) :: comm_graph

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes. If the size, nnodes, of the graph is smaller than the size of the group of [comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments **nnodes**, **index**, and **edges** are illustrated with the following simple example.

Example 7.2

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors	
0	1, 3	
1	0	
2	3	
3	0, 2	

Then, the input arguments are:

nnodes =	4
index =	2, 3, 4, 6
edges =	1,3,0,3,0,2

Thus, in C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges[j], for $0 \le j \le index[0] - 1$ and the list of neighbors of node i, i > 0, is stored in edges[j], index[i-1] $\le j \le index[i] - 1$.

In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in 47 edges(j), for $1 \le j \le index(1)$ and the list of neighbors of node i, i > 0, is stored in 48

Unofficial Draft for Comment Only

 24

 $_{15}$ ticket0.

1 $edges(j), index(i) + 1 \le j \le index(i + 1).$ $\mathbf{2}$ A single process is allowed to be defined multiple times in the list of neighbors of a 3 process (i.e., there may be multiple edges between two processes). A process is also allowed 4 to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed 5to be non-symmetric. 6 Advice to users. Performance implications of using multiple edges or a non-symmetric 7 adjacency matrix are not defined. The definition of a node-neighbor edge does not 8 imply a direction of the communication. (End of advice to users.) 9 10 Advice to implementors. The following topology information is likely to be stored 11 with a communicator: 12• Type of topology (Cartesian/graph), 13 14• For a Cartesian topology: 151. ndims (number of dimensions), 162. dims (numbers of processes per coordinate direction), 17 3. periods (periodicity information), 18 4. own_position (own position in grid, could also be computed from rank and 19 dims) 2021• For a graph topology: 22 1. index. 232. edges, 24which are the vectors defining the graph structure. 2526For a graph structure the number of nodes is equal to the number of processes in 27the group. Therefore, the number of nodes does not have to be stored explicitly. 28An additional zero entry at the start of array index simplifies access to the topology 29 information. (End of advice to implementors.) 30 31 ticket259. $_{32}$ 7.5.4 [Distributed (Graph)]Distributed Graph Constructor ticket259. ₃₃ [The general graph constructor assumes]MPI_GRAPH_CREATE requires that each process 34 passes the full (global) communication graph to the call. This limits the scalability of this 35 constructor. With the distributed graph interface, the communication graph is specified 36 in a fully distributed fashion. Each process specifies only the part of the communication 37 graph of which it is aware. Typically, this could be the set of processes from which the 38 process will eventually receive or get data, or the set of processes to which the process will 39 send or put data, or some combination of such edges. Two different interfaces can be used 40to create a distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a ticket0.⁴¹ distributed graph communicator with each process specifying [all]each of its incoming and 42outgoing (adjacent) edges in the logical communication graph and thus requires minimal ticket259. 43 communication during creation. [MPI_DIST_GRAPH_CREATE provides full flexibility, and 44processes can indicate that communication will occur between other pairs of processes. 45MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that 46communication will occur between any pair of processes in the graph. 47To provide better possibilities for optimization by the MPI library, the distributed 48graph constructors permit weighted communication edges and take an info argument that

7.5. TOPOLOGY CONSTRUCTORS

can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

```
MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, out-
degree, destinations, destweights, info, reorder, comm_dist_graph)
```

	IN	comm_old	input communicator (handle)	8
	IN	indegree	size of sources and sourceweights arrays (non-negative	9
		indegree	integer)	10
			- ,	11
	IN	sources	ranks of processes for which the calling process is a	12
			destination (array of non-negative integers)	13
	IN	sourceweights	weights of the edges into the calling process (array of	14
			non-negative integers)	15
	IN	outdegree	size of destinations and destweights arrays (non-negative	16
		outdegree	integer)	17
		1	- ,	18
	IN	destinations	ranks of processes for which the calling process is a	19
			source (array of non-negative integers)	20
	IN	destweights	weights of the edges out of the calling process (array	21
			of non-negative integers)	22
	IN	info	hints on optimization and interpretation of weights	23
			(handle)	24
				25
	IN	reorder	the ranks may be reordered (true) or not (false) (logi-	26
			cal)	27
	OUT	comm_dist_graph	communicator with distributed graph topology (han-	28
			dle)	29
				30
i	nt MPI_Di	.st_graph_create_adjacent	(MPI_Comm comm_old, int indegree, const	31 ticket 140.
		<pre>int sources[], const</pre>	<pre>int sourceweights[], int outdegree, const</pre>	$^{32}_{22}$ ticket 140.

 33 ticket 140.

 34 ticket 140.

³⁵ ticket 229.2.

³⁶ ticket-248T.

1	MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
2	OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
3 4	COMM_DIST_GRAPH, IERROR)
5	<pre>INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR</pre>
6	LOGICAL REORDER
7	
8	{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int
9	<pre>indegree, const int sources[], const int sourceweights[],</pre>
10	<pre>int outdegree, const int destinations[],</pre>
11	<pre>const int destweights[], const MPI::Info& info, bool reorder) const(binding deprecated, see Section 15.2) }</pre>
12	const (binaing deprecated, see Section 15.2) }
13	$\{\texttt{MPI::Distgraphcomm}$
14	<pre>MPI::Intracomm::Dist_graph_create_adjacent(int indegree,</pre>
15 16	<pre>const int sources[], int outdegree, const int destinations[],</pre>
10	const MPI::Info& info, bool reorder) const(binding deprecated, see
18	Section 15.2 }
19	MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator to
ticket 259. $_{20}$	which the distributed graph topology information is attached. [Each process passes all
21	information about the edges to its neighbors]Each process passes all information about its
22	incoming and outgoing edges in the virtual distributed graph topology. The calling processes
23	must ensure that each edge of the graph is described in the source and in the destination
24	process with the same weights. If there are multiple edges for a given (source,dest) pair, then the sequence of the weights of these edges does not matter. The complete communication
25	topology is the combination of all edges shown in the sources arrays of all processes in
26	comm_old, which must be identical to the combination of all edges shown in the destinations
27 28	arrays. Source and destination ranks must be process ranks of comm_old. This allows a fully
29	distributed specification of the communication graph. Isolated processes (i.e., processes with
30	no outgoing or incoming edges, that is, processes that have specified indegree and outdegree
ticket 259. $_{31}$	as zero and [that]thus do not occur as source or destination rank in the graph specification)
32	are allowed.
33	The call creates a new communicator comm_dist_graph of distributed graph topology
34	type to which topology information has been attached. The number of processes in
35	comm_dist_graph is identical to the number of processes in comm_old. The call to
36	MPI_DIST_GRAPH_CREATE_ADJACENT is collective. Weights are specified as non-negative integers and can be used to influence the process
37	remapping strategy and other internal MPI optimizations. For instance, approximate count
38	arguments of later communication calls along specific edges could be used as their edge
39 40	weights. Multiplicity of edges can likewise indicate more intense communication between
40	pairs of processes. However, the exact meaning of edge weights is not specified by the MPI
42	standard and is left to the implementation. In C or Fortran, an application can supply
43	the special value $MPI_UNWEIGHTED$ for the weight array to indicate that all edges have the
44	same (effectively no) weight. In C++, this constant does not exist and the weight arguments
45	may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or
46	in C++ omit the weight arrays, for some but not all processes of comm_old. Note that
47	MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total
48	array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is an

object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

The meaning of the **info** and **reorder** arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n, set	sources, degrees,	destinations,	weights,	info,	re-
order, comm_dist_graph)					

	ст,							
IN	comm_old	input communicator (handle)	8					
IN	n	number of source nodes for which this process specifies	9					
		edges (non-negative integer)	10					
IN	sources	array containing the n source nodes for which this pro-	11 12					
	3001003	cess specifies edges (array of non-negative integers)	12					
IN	dogroop		14					
IIN	degrees	array specifying the number of destinations for each source node in the source node array (array of non-	15					
		negative integers)	16					
INI	destinations	,	17					
IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)	18					
			19					
IN	weights	weights for source to destination edges (array of non-	20					
		negative integers)	21					
IN	info	hints on optimization and interpretation of weights	22					
		(handle)	23					
IN	reorder	the process may be reordered $(true)$ or not $(false)$ (log-	24 25					
		ical)	25					
OUT	comm_dist_graph	communicator with distributed graph topology added	27					
		(handle)	28					
			29					
int MPI_	Dist_graph_create(MPI_Com	<pre>m comm_old, int n, const int sources[],</pre>	³⁰ ticket140.					
	<pre>const int degrees[],</pre>	<pre>const int destinations[], const</pre>	31 ticket 140.					
	int weights[], MPI_I	nfo info, int reorder,	$_{32}$ ticket 140.					
	MPI_Comm *comm_dist_	graph)	33 ticket140.					
MPT Dist	graph create(comm old n	, sources, degrees, destinations, weights,	$_{34}$ ticket-248T.					
		dist_graph, ierror) BIND(C)	35					
TYPE	(MPI_Comm), INTENT(IN) ::		36					
		urces(n), degrees(n), destinations(*)	37					
INTE	GER, INTENT(IN) :: weigh	ts(*)	38 39					
TYPE	(MPI_Info), INTENT(IN) ::	info	40					
LOGI		41						
	(MPI_Comm), INTENT(OUT) :	U	42					
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	43					
MPI_DIST	_GRAPH_CREATE(COMM_OLD, N	, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	44					
	INFO, REORDER, COMM_		45					
INTE	GER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),	46					
WEIG	HTS(*), INFO, COMM_DIST_G	RAPH, IERROR	47					

LOGICAL REORDER

1

 2

```
312
```

```
1
2
```

3

4

5

6

 $\overline{7}$

8

9

{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,

const int sources[], const int degrees[], const int

destinations[], const int weights[], const MPI::Info& info,

```
bool reorder) const(binding deprecated, see Section 15.2) }
```

{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,

const int sources[], const int degrees[],

const int destinations[], const MPI::Info& info, bool reorder)
const(binding deprecated, see Section 15.2) }

10 MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the 11distributed graph topology information is attached. Concretely, each process calls the con-12structor with a set of directed (source, destination) communication edges as described below. 13Every process passes an array of n source nodes in the sources array. For each source node, a 14non-negative number of destination nodes is specified in the degrees array. The destination 15nodes are stored in the corresponding consecutive segment of the destinations array. More 16precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the i-th 17such edge stored in destinations[degrees[0]+...+degrees[i-1]+i]. The weight of this edge is 18 stored in weights [degrees[0]+...+degrees[i-1]+j]. Both the sources and the destinations arrays 19may contain the same node more than once, and the order in which nodes are listed as 20destinations or sources is not significant. Similarly, different processes may specify edges 21with the same source and destination nodes. Source and destination nodes must be pro-22cess ranks of comm_old. Different processes may specify different numbers of source and 23destination nodes, as well as different source to destination edges. This allows a fully dis- 24 tributed specification of the communication graph. Isolated processes (i.e., processes with 25no outgoing or incoming edges, that is, processes that do not occur as source or destination 26node in the graph specification) are allowed.

ticket259.

$t259._{30}^{29}$

27

28

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to [MPI_Dist_graph_create]MPI_DIST_GRAPH_CREATE is collective.

³¹ If reorder = false, all processes will have the same rank in comm_dist_graph as in ³² comm_old. If reorder = true then the MPI library is free to remap to other processes (of ³³ comm_old) in order to improve communication on the edges of the communication graph. ³⁴ The weight associated with each edge is a hint to the MPI library about the amount or ³⁵ intensity of communication on that edge, and may be used to compute a "best" reordering.

36 Weights are specified as non-negative integers and can be used to influence the process 37 remapping strategy and other internal MPI optimizations. For instance, approximate count 38 arguments of later communication calls along specific edges could be used as their edge 39 weights. Multiplicity of edges can likewise indicate more intense communication between 40pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 41 standard and is left to the implementation. In C or Fortran, an application can supply 42the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 43same (effectively no) weight. In C++, this constant does not exist and the weights argument 44may be omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or 45in C++ omit the weight arrays, for some but not all processes of comm_old. Note that 46MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total 47array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is 48

an object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors*.)

Example 7.3 As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	$0,\!1,\!2,\!3$	2,1,1,2	1,3,0,3,0,2	$1,\!1,\!1,\!1,\!1,\!1,\!1$
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

In both cases above, the application could supply MPI_UNWEIGHTED instead of explic-

Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8

9

10

11

12 13

14

15

16

17

18

19

20 21

22

 31

32

40

```
itly providing identical weights.
       \mathbf{2}
                 MPI_DIST_GRAPH_CREATE_ADJACENT could be used to specify this graph using the
       3
             following arguments:
       4
                                                                                       destweights
                   process
                            indegree
                                      sources
                                               sourceweights
                                                              outdegree
                                                                         destinations
       5
                      0
                            \mathbf{2}
                                      1.3
                                               ^{1,1}
                                                              \mathbf{2}
                                                                          1.3
                                                                                       1,1
       6
                                                              1
                                                                                       1
                      1
                            1
                                      0
                                               1
                                                                         0
       7
                      2
                                               1
                                                              1
                                                                         3
                                                                                       1
                            1
                                      3
       8
                                                              2
                      3
                            \mathbf{2}
                                      0.2
                                                                         0.2
                                               1,1
                                                                                       1.1
       9
       10
       11
             Example 7.4 A two-dimensional PxQ torus where all processes communicate along the
ticket
0. ^{\scriptscriptstyle 12}
             dimensions and along the diagonal edges. This cannot be modelled modeled with Cartesian
       13
             topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the
       14
             following code. In this example, the communication along the dimensions is twice as heavy
       15
             as the communication along the diagonals:
       16
       17
             /*
       ^{18}
             Input:
                         dimensions P, Q
       19
             Condition: number of processes equal to P*Q; otherwise only
       20
                         ranks smaller than P*Q participate
       21
             */
       22
             int rank, x, y;
       23
             int sources[1], degrees[1];
       ^{24}
             int destinations[8], weights[8];
       25
            MPI_Comm comm_dist_graph;
       26
       27
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       28
       29
             /* get x and y dimension */
       30
            y=rank/P; x=rank%P;
       31
       32
             /* get my communication partners along x dimension */
       33
             destinations[0] = P*y+(x+1)%P; weights[0] = 2;
       34
             destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
       35
       36
             /* get my communication partners along y dimension */
       37
             destinations[2] = P*((y+1))(Q)+x; weights[2] = 2;
       38
             destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
       39
       40
             /* get my communication partners along diagonals */
       41
             destinations[4] = P*((y+1))(Q)+(x+1)(P); weights[4] = 1;
       42
             destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
       43
             destinations[6] = P*((y+1))(Q) + (P+x-1)(P); weights[6] = 1;
       44
             destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
       45
       46
             sources[0] = rank;
       47
             degrees [0] = 8;
       48
             MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
```

	weig	<pre>hts, MPI_INFO_NULL, 1, &comm_dist_graph);</pre>	1		
7.5.5 To	pology Inquiry Funct	ions	3		
			4		
	01	ith one of the above functions, then the topology information	5		
can be loo	ked up using inquiry	functions. They all are local calls.	6		
			7		
	O_TEST(comm, statı		8		
	O_TEST(comm, statt		9		
IN	comm	$\operatorname{communicator}$ (handle)	10		
OUT	status	topology type of communicator comm (state)	11		
			12		
int MPT 1	Topo test(MPT Comm	comm, int *status)	13		
	-		14 ticket-248T.		
-	_test(comm, status		15		
	(MPI_Comm), INTENT		16		
	GER, INTENT(OUT) :		17		
INTEC	GER, OPTIONAL, INT	ENT(OUT) :: ierror	18		
MPI TOPO	_TEST(COMM, STATUS	. IERROR)	19		
	GER COMM, STATUS,		20		
			21		
{int MPI:	::Comm::Get_topolo	gy() const(binding deprecated, see Section 15.2) }	22		
The f	function MPI TOPO	_TEST returns the type of topology that is assigned to a	23		
communic			24 25		
The output value status is one of the following:					
	1	0	26		
MPI_GRA	APH	graph topology	27		
MPI_CAI	RT	Cartesian topology	28		
	T_GRAPH	distributed graph topology	29		
MPI_UN	DEFINED	no topology	30		
			31 32		
			32		
MPI_GRA	PHDIMS_GET(comm	, nnodes, nedges)	34		
IN	comm	communicator for group with graph structure (handle)	35		
OUT	nnodes	number of nodes in graph (integer) (same as number	36		
001	modes	of processes in the group)	37		
0.U.T		/	38		
OUT	nedges	number of edges in graph (integer)	39		
			40		
int MPI_(<pre>Fraphdims_get(MPI_</pre>	Comm comm, int *nnodes, int *nedges)	41 ticket-248T.		
MPI_Graph	ndims_get(comm, nn	odes, nedges, ierror) BIND(C)	42		
-	(MPI_Comm), INTENT	e	43		
	GER, INTENT(OUT) :		44		
		ENT(OUT) :: ierror	45		
			46		
	-	ODES, NEDGES, IERROR)	47		
INTEC	GER COMM, NNODES,	NEDGED, IEKKUK	48		

		316		CHAPTER 7. PROCESS TOPOLOGIES		
	1 2	{void MPI:	:Graphcomm::Get_dims(int deprecated, see Section 1	<pre>nnodes[], int nedges[]) const(binding 5.2) }</pre>		
	3 4 5 6 7	Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-top information that was associated with a communicator by MPI_GRAPH_CREATE. The information provided by MPI_GRAPHDIMS_GET can be used to dimensio vectors index and edges correctly for the following call to MPI_GRAPH_GET				
	8 9	MPI GRAPI	H_GET(comm, maxindex, max	(edges, index, edges)		
	10 11	IN	comm	communicator with graph structure (handle)		
	12 13	IN	maxindex	length of vector index in the calling program (integer)		
	14 15	IN	maxedges	length of vector edges in the calling program (integer)		
	16 17 18	OUT	index	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)		
	19	OUT	edges	array of integers containing the graph structure		
ticket125. ticket126.		<pre>int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,</pre>				
ticket-248T.		<pre>MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: maxindex, maxedges INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges) INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
	29 30	MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR				
	31 32 33 34	<pre>{void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[],</pre>				
	35 36	MPI_CARTI	DIM_GET(comm, ndims)			
	37 38	IN	comm	communicator with Cartesian structure (handle)		
	39 40	OUT	ndims	number of dimensions of the Cartesian structure (in- teger)		
	41 42	int MPI_Ca	artdim_get(MPI_Comm comm,	int *ndims)		
ticket-248T.	43 44 45 46 47	<pre>MPI_Cartdim_get(comm, ndims, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(OUT) :: ndims INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
	48	MPI_CARTDI	M_GET(COMM, NDIMS, IERRO	R)		

TNTEG	ER COMM, NDIMS, IERROR		1	
		<pre>st(binding deprecated, see Section 15.2) }</pre>	2	
			3	
		and MPI_CART_GET return the Cartesian topol-	4	
0.0		a communicator by MPI_CART_CREATE. If comm	5	
		Cartesian topology, MPI_CARTDIM_GET returns	7	
ndims=0 ar	IG MIPI_CART_GET WIII Keep	all output arguments unchanged.	8	
			9	
MPI_CART	_GET(comm, maxdims, dims,	periods, coords)	10	
IN	comm	communicator with Cartesian structure (handle)	11	
IN	maxdims	length of vectors dims, periods, and coords in the	12	
	maxamis	calling program (integer)	13	
OUT	dims	number of processes for each Cartesian dimension (ar-	14	
001	Ginio	ray of integer)	15 16	
	periods		17	
OUT	penous	periodicity (true/false) for each Cartesian dimension (array of logical)	18	
	ee erde		19	
OUT	coords	coordinates of calling process in Cartesian structure (array of integer)	20	
		(array of integer)	21	
int MDT C	nt maxdims, int [*dims]dims[],	22 tieleot 195		
IIIC MPI_Co	-	s[], int [*coords]coords[])	$_{23}$ ticket125. $_{24}$ ticket125.	
			ticket125	
MPI_Cart_{	$^{25}_{26}$ ticket-248T.			
TYPE()	20			
	ER, INTENT(IN) :: maxdin	ns (maxdims), coords(maxdims)	28	
	AL, INTENT(OUT) :: perio		29	
INTEGI	30			
			31	
		PERIODS, COORDS, IERROR)	32	
	ER COMM, MAXDIMS, DIMS(*)), CUURDS(*), IERRUR	33	
	AL PERIODS(*)		34	
$\{void MPI\}$		<pre>maxdims, int dims[], bool periods[],</pre>	35	
	<pre>int coords[]) const()</pre>	binding deprecated, see Section 15.2) }	36 37	
			37 38	
			39	
MPI_CART	40			
IN	comm	communicator with Cartesian structure (handle)	41	
IN	coords	integer array (of size ndims) specifying the Cartesian	42	
		coordinates of a process	43	
OUT	rank	rank of specified process (integer)	44	
			45	
int MPI Ca	<pre>const int [*coords]coords[], int *rank)</pre>	$_{47}^{46}$ ticket 140.		
MPI_Cart_1	rank(comm, coords, rank,	lerror) BIND(C)	48 ticket-248T.	

	1 2 3 4	INTEGE INTEGE	MPI_Comm), INTENT(IN) ER, INTENT(IN) :: coo ER, INTENT(OUT) :: ra ER, OPTIONAL, INTENT(O	rds(*) nk			
	5 6 7	MPI_CART_RANK(COMM, COORDS, RANK, IERROR) INTEGER COMM, COORDS(*), RANK, IERROR					
	8 9 10	<pre>{int MPI::Cartcomm::Get_cart_rank(const int coords[]) const(binding</pre>					
	11 12 13	lates the log routines.	For dimension i with periods(i) = true, if the coordinate, coords(i), is our				
	14For dimension 1 with periods(1) = true, if the coordinate, coords(1),15range, that is, coords(i) < 0 or coords(i) \geq dims(i), it is shifted back to th16 $0 \leq$ coords(i) < dims(i) automatically. Out-of-range coordinates are error						
	19 20 21		is returned in rank.				
	21	MPI_CART	_COORDS(comm, rank, m	axdims, coords)			
	23	IN	comm	communicator with Cartesian structure (handle)			
	24	IN	rank	rank of a process within group of $comm$ (integer)			
	25 26 27	IN	maxdims	length of vector $coords$ in the calling program (integer)			
	28 29 30	OUT	coords	integer array (of size ndims) containing the Cartesian coordinates of specified process (array of integers)			
ticket125.		<pre>int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,</pre>					
ticket-248T.	33 34 35 36 37 38	<pre>MPI_Cart_coords(comm, rank, maxdims, coords, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: rank, maxdims INTEGER, INTENT(OUT) :: coords(maxdims) INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
	39 40	MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR) INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR					
	41 42 43	$\{void MPI:$		<pre>(int rank, int maxdims, int coords[]) ated, see Section 15.2) }</pre>			
	44 45 46 47 48	The inverse mapping, rank-to-coordinates translation is provided by MPI_CART_COORDS. If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.					

MPI_GRA	PH_NEIGHBORS_COUNT(co	mm, rank, nneighbors)	1
IN	comm	communicator with graph topology (handle)	2
IN	rank	rank of process in group of comm (integer)	3
OUT	nneighbors	number of neighbors of specified process (integer)	4 5
001	Intelginors	number of neighbors of specified process (integer)	6
int MDT (Tranh naighbarg count (MDI	[_Comm comm, int rank, int *nneighbors)	7
IIIC MFI_(apii_iieigiiboi s_couiic(Mr)	L_COMM COMM, INC TAIK, INC *INTERGROOTS)	8 ticket-248T.
_		cank, nneighbors, ierror) BIND(C)	9
	(MPI_Comm), INTENT(IN) ::	Comm	10
	GER, INTENT(IN) :: rank GER, INTENT(OUT) :: nnei	ichhora	11
	GER, OPTIONAL, INTENT(OUT		12 13
			14
		AANK, NNEIGHBORS, IERROR)	15
INTEC	GER COMM, RANK, NNEIGHBOF	RS, IERRUR	16
$\{\texttt{int MPI}\}$		cs_count(int rank) const(binding deprecated,	17
	see Section 15.2 }		18
			19
		· · · · · · · · · · · · · · · · · · ·	20
MPI_GRA	PH_NEIGHBORS(comm, rank	c c ,	21 22
IN	comm	communicator with graph topology (handle)	23
IN	rank	rank of process in group of comm (integer)	24
IN	maxneighbors	size of array neighbors (integer)	25
OUT	neighbors	ranks of processes that are neighbors to specified pro-	26
001	neignooro	cess (array of integer)	27
			28
int MPI_(Graph_neighbors(MPI_Comm	comm, int rank, int maxneighbors,	29 30
	int [*neighbors]neig	ghbors[])	$_{31}$ ticket 125.
MPT Grant	neighbors(comm rank m	maxneighbors, neighbors, ierror) BIND(C)	$_{32}$ ticket-248T.
-	(MPI_Comm), INTENT(IN) ::		33
	GER, INTENT(IN) :: rank,		34
INTE	GER, INTENT(OUT) :: neig	ghbors(maxneighbors)	35
INTE	GER, OPTIONAL, INTENT(OUT	[) :: ierror	36
MPI_GRAPH	H_NEIGHBORS(COMM, RANK, M	MAXNEIGHBORS, NEIGHBORS, IERROR)	37 38
		BORS, NEIGHBORS(*), IERROR	39
{void MP	ors(int rank, int maxneighbors, int	40	
ניטבע ווו	41		
	-	binding deprecated, see Section 15.2) }	42
		T and MPI_GRAPH_NEIGHBORS provide adjacency ogy. The returned count and array of neighbors for	$^{43}_{44}{ m ticket}259.$
	10 10 1	by The returned count and array of heighbors for l neighbors and reflect the same edge ordering as	44 UCKet239.
-		MPI_GRAPH_CREATE. Specifically,	45
-		d MPI_GRAPH_NEIGHBORS will return values based	47
			48

on the original index and edges array passed to MPI_GRAPH_CREATE (assuming that index[-1] effectively equals zero):

• The [count]number of neighbors (nneighbors) returned from MPI_GRAPH_NEIGHBORS_COUNT will be (index[rank] - index[rank-1]).

• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-1]] through edges[index[rank]-1].

Example 7.5

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

process	neighbors
0	1, 1, 3
1	0, 0
2	3
3	0, 2, 2

Thus, the input arguments to MPI_GRAPH_CREATE are:

nnodes =4 index =3, 5, 6, 9 1, 1, 3, 0, 0, 3, 0, 2, 2edges =

Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for each of the 4 processes will return:

	Input rank	Count	Neighbors
_	0	3	1, 1, 3
	1	2	0, 0
	2	1	3
	3	3	0, 2, 2
-			

Example 7.6

35 Suppose that **comm** is a communicator with a shuffle-exchange topology. The group has 36 2^n members. Each process is labeled by a_1, \ldots, a_n with $a_i \in \{0, 1\}$, and has three neighbors: 37 exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n \ (\bar{a} = 1 - a), \text{ shuffle}(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1, \text{ and}$ 38unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for 39 n = 3.

- 40 41
- 42
- 43
- 44
- 45
- 46
- 4748

1

 2

3

56

7

8 9 10

11

12

19

2021

22

23

 24

2526

33 34

ticket0. 4

node		exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator comm has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
C assume: each process has stored a real number A.
                                                                                  16
С
 extract neighborhood information
                                                                                  17
      CALL MPI_COMM_RANK(comm, myrank, ierr)
                                                                                  18
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
                                                                                  19
C perform exchange permutation
                                                                                  20
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
                                                                                  21
     +
           neighbors(1), 0, comm, status, ierr)
                                                                                  22
C perform shuffle permutation
                                                                                  23
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
                                                                                  24
           neighbors(3), 0, comm, status, ierr)
                                                                                  25
C perform unshuffle permutation
                                                                                  26
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
                                                                                  27
     +
           neighbors(2), 0, comm, status, ierr)
                                                                                  28
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS provide adjacency information for a distributed graph topology.

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)

	IN	comm	communicator with distributed graph topology (han-	34
	IIN	comm	dle)	35
				36
	OUT	indegree	number of edges into this process (non-negative inte-	37
			ger)	38
	OUT	outdegree	number of edges out of this process (non-negative in-	39
	-	teger)	40	
	OUT	weighted	false if MPI_UNWEIGHTED was supplied during cre-	41
001	weighted	ation, true otherwise (logical)	42	
		ation, the otherwise (logical)	43	
				44

MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
BIND(C)

Unofficial Draft for Comment Only

12

13

14 15

29

30

31 32 33

24

45

47

48

 46 ticket-248T.

	1 2 3 4 5 6 7 8	INTEG LOGIC INTEG MPI_DIST_ INTEG	MPI_Comm), INTENT(IN) :: ER, INTENT(OUT) :: indeg AL, INTENT(OUT) :: weigh ER, OPTIONAL, INTENT(OUT) GRAPH_NEIGHBORS_COUNT(COM ER COMM, INDEGREE, OUTDEC AL WEIGHTED	gree, outdegree nted) :: ierror MM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)		
	9 10 11 12 13	<pre>{void MPI::Distgraphcomm::Get_dist_neighbors_count(int rank,</pre>				
	14 15	MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights)				
	16 17 18	IN	comm	communicator with distributed graph topology (han-dle)		
	19 20	IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)		
	21 22 23	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)		
	23 24 25	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)		
	26 27	IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)		
	28 29 30	OUT	destinations	processes for which the calling process is a source (ar- ray of non-negative integers)		
	31 32	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)		
ticket229.2.	33 34 35 36	<pre>int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>				
ticket-248T.		<pre>MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm</pre>				
	41 42 43 44	<pre>INTEGER, INTENT(IN) :: maxindegree, maxoutdegree INTEGER, INTENT(OUT) :: sources(maxindegree), destinations(maxoutdegree) INTEGER :: sourceweights(*), destweights(*)</pre>				
	45 46 47 48	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,				

MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. The If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in **comm_old** in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBOR_COUNT, then only the first part of the full list is returned. [Note, that the order of returned edges does need not to be identical to the order that was provided in the creation of comm for the case that MPI_DIST_GRAPH_CREATE_ADJACENT was used.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

```
14

<sup>15</sup> ticket258.

16

17

18

19
```

8

9

10

11

12

13

20

21

 24

25 26

27

28

29

30 31 32

33

34

35

36

37

38

39

²² ticket258. ²³

	1	MPI_CART	_SHIFT(comm, direction	n, disp, rank_source, rank_dest)			
	2	IN	comm	communicator with Cartesian structure (handle)			
	3 4	IN	direction	coordinate dimension of shift (integer)			
	5	IN	disp	displacement (> 0: upwards shift, < 0 : downwards			
	6		alsp	shift) (integer)			
	7	OUT	rank_source	rank of source process (integer)			
	8 9	OUT	- rank_dest	rank of destination process (integer)			
	9 10	001		rum of dependence process (medger)			
	11	int MPI_Ca	rt_shift(MPI_Comm c	comm, int direction, int disp,			
· 1 · 0.40m	12	<pre>int *rank_source, int *rank_dest)</pre>					
ticket-248T	• 13	MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)					
	14	BIND(C)					
	15 16	TYPE(MPI_Comm), INTENT(IN) :: comm					
	17	INTEGE	CR, INTENT(IN) :: d	irection, disp			
	18			rank_source, rank_dest			
	19	INTEGE	R, OPTIONAL, INTENT	'(OUT) :: ierror			
	20	MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)					
	21	INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR					
	22 23	<pre>{void MPI::Cartcomm::Shift(int direction, int disp, int& rank_source,</pre>					
	23 24	<pre>int& rank_dest) const(binding deprecated, see Section 15.2) }</pre>					
ticket0	. 25	The [di	irection]direction argum	nent indicates the coordinate dimension to be traversed			
	26			umbered from 0 to $ndims-1$, where $ndims$ is the number			
	27	of dimensions. Depending on the periodicity of the Cartesian group in the specified coordinate direc-					
	28 29	tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case					
	30	,	-	ROC_NULL may be returned in rank_source or rank_dest,			
	31	indicating that the source or the destination for the shift is out of range.					
	32	It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or					
	33	greater than or equal to the number of dimensions in the Cartesian communicator. This					
	34	implies that it is erroneous to call MPI_CART_SHIFT with a $comm$ that is associated with					
	35	a zero-dime	nsional Cartesian topol	ogy.			
	36	Europealo 5	7 7				
	37 38	Example 7		s a two-dimensional, periodic, Cartesian topology associ-			
	38 39		, , ,	ray of REALs is stored one element per process, in variable			
	40			by shifting column i (vertically, i.e., along the column)			
	41	by i steps.	ies to skew tills array,	by sinting column 1 (vertically, i.e., along the column)			
	42	by i bteps.					
	43						
	44	C find pro	cess rank				
	45		. MPI_COMM_RANK(comm	n, rank, ierr)			
	46		tesian coordinates				
	47			mm, rank, maxdims, coords, ierr)			
	48	C compute	shift source and de	stination			

CHAPTER 7. PROCESS TOPOLOGIES

C skew a	rray LL MPI_SENDRECV_REPLACE(A	D, coords(2), source, dest, ierr) A, 1, MPI_REAL, dest, 0, source, 0, comm, status, ierr)	1 2 3 4		
Advice to users. In Fortran, the dimension indicated by $DIRECTION = i$ has $DIMS(i+1)$ nodes, where $DIMS$ is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (End of advice to users.)					
7.5.7 Partitioning of Cartesian [s]Structures					
MPL CAR	RT_SUB(comm, remain_dims, ı	newcomm)	12 13		
IN IN	· ·	communicator with Cartesian structure (handle)	14		
	comm		15		
IN	remain_dims	the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is drop-	16 17		
		ped (false) (logical vector)	18		
OUT	newcomm	communicator containing the subgrid that includes	19		
001	newcomm	the calling process (handle)	20		
			21		
int MPI_	Cart_sub(MPI_Comm comm, o	<pre>const int [*remain_dims]remain_dims[],</pre>	$^{22}_{23}$ ticket140.		
	MPI_Comm *newcomm)		26 ticket126.		
MPI_Cart	_sub(comm, remain_dims, 1	newcomm, ierror) BIND(C)	ticket-248T. 25		
TYPE	26				
LOGI	27				
TYPE	28 29				
INTE	30				
	_SUB(COMM, REMAIN_DIMS, I		31		
	GER COMM, NEWCOMM, IERRO	R	32		
LUGI	CAL REMAIN_DIMS(*)		33		
{MPI::Ca		<pre>o(const bool remain_dims[]) const(binding</pre>	34		
	deprecated, see Section	15.2) }	35		
If a	Cartesian topology has been	n created with MPI_CART_CREATE, the function	36 37		
		ition the communicator group into subgroups that	38		
	0	rids, and to build for each subgroup a communicator	39		
	_	topology. If all entries in remain_dims are false or	40		
	-	o-dimensional Cartesian topology then newcomm is	41		
		rtesian topology. (This function is closely related to	42 43		
	MPI_COMM_SPLIT.)				
Example			44 45		
Assu	45				
remain_d	47				
MPI	48				

will create three communicators each with eight processes in a 2 × 4 Cartesian topology. If remain_dims = (false, false, true) then the call to MPI_CART_SUB(comm, remain_dims, comm_new) will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

7.5.8 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI. The two calls are both local.

```
MPI_CART_MAP(comm, ndims, dims, periods, newrank)
```

	15	IN	comm	input communicator (handle)			
	16	IN	ndims	number of dimensions of Cartesian structure (integer)			
	17 18	IN	dims	integer array of size ndims specifying the number of processes in each coordinate direction			
	19 20 21	IN	periods	logical array of size ndims specifying the periodicity specification in each coordinate direction			
	22 23 24 25	OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)			
ticket140. ticket126. ticket140	27	<pre>int MPI_Cart_map(MPI_Comm comm, int ndims, const int [*dims]dims[], const</pre>					
ticket-248T		<pre>MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: ndims, dims(ndims) LOGICAL, INTENT(IN) :: periods(ndims) INTEGER, INTENT(OUT) :: newrank INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
	35 36 37 38	MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR LOGICAL PERIODS(*)					
	39 40	{int MPI::	-	<pre>t ndims, const int dims[], const bool periods[]) deprecated, see Section 15.2) }</pre>			
	41 42 43 44	ical machine	e. A possible impler	s an "optimal" placement for the calling process on the phys- nentation of this function is to always return the rank of the perform any reordering.			
	45 46 47 48	Advice to implementors. The function MPI_CART_CREATE(comm, ndims, dims, riods, reorder, comm_cart), with reorder = true can be implemented by calling MPI_CART_MAP(comm, ndims, dims, periods, newrank), then call MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank					

1

 2

3

4

5 6

 $\overline{7}$

 $12 \\ 13$

	$^{1}_{2}$ ticket162.					
	3 4 5 6 7					
	All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (<i>End of advice to implementors.</i>)					
r	The corresponding [new]fur	action for [general]graph structures is as follows.	$^{10}_{11} ext{ticket259.} \ _{12} ext{ticket259.}$			
MPI_	GRAPH_MAP(comm, nnode	es, index, edges, newrank)	13			
IN	comm	input communicator (handle)	14			
IN	nnodes	number of graph nodes (integer)	15 16			
			17			
IN	index	integer array specifying the graph structure, see MPI_GRAPH_CREATE	18			
IN	edges	integer array specifying the graph structure	19			
OU	-	reordered rank of the calling process;	20 21			
00		MPI_UNDEFINED if the calling process does not be-	22			
		long to graph (integer)	23			
			24			
int 1		<pre>comm, int nnodes, const int [*index]index[], ges]edges[], int *newrank)</pre>	²⁵ ticket140. ²⁶ ticket126.			
MPI_	 ²⁷ ticket140. ²⁸ ticket126. ²⁹ ticket-248T. 					
	-	<pre>nnodes, index(nnodes), edges(*)</pre>	30			
	INTEGER, INTENT(OUT) ::		31			
	INTEGER, OPTIONAL, INTE	NT(UUT) :: lerror	32			
MPI_	GRAPH_MAP(COMM, NNODES,	INDEX, EDGES, NEWRANK, IERROR)	33			
	INTEGER COMM, NNODES, I	NDEX(*), EDGES(*), NEWRANK, IERROR	34 35			
{int	MPI::Graphcomm::Map(int	t nnodes, const int index[], const int edges[])	36			
(eprecated, see Section 15.2 }	37			
			38			
	Advice to implementors	The function MPI_GRAPH_CREATE(comm, nnodes, index,	39			
	-), with reorder = true can be implemented by calling	40			
	MPI_GRAPH_MAP(comm,		41			
		color, key, comm_graph), with color = 0 if newrank \neq	42			
	MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.					
All other graph topology functions can be implemented locally, using the topology						
information that is cached with the communicator. (End of advice to implementors.)						
	$^{46}_{47}$ ticket258.					

1 2

3

4

5

6

7

8

9

17

18

19

20

21

22

23

7.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 5 on page 151 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources and destinations containing 0, ..., n-1, where n is the number of processes in the group of comm_old (i.e., the graph is fully connected and includes also an edge from each node to itself), then the sparse neighborhood communication routine performs the same data exchange as the corresponding dense (fully-connected) collective operation. In the case of a Cartesian communicator, only nearest neighbor communication is provided, corresponding to rank_source and rank_dest in MPI_CART_SHIFT with input disp=1.

Rationale. Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [35]. This functionality can significantly simplify the implementation of neighbor exchanges [31]. (End of rationale.)

For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined as the sequence returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For a general graph topology, created with MPI_GRAPH_CREATE, the order of neighbors in the send and receive buffers is defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that general graph topologies should generally be replaced by the distributed graph topologies.

For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined by order of the dimensions, first the neighbor in the negative direction and then in the positive direction with displacement 1. The numbers of sources and destinations in the communication routines are **2*ndims** with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., **periods[...]==false**), then this neighbor is defined to be MPI_PROC_NULL.

If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI_PROC_NULL in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

42 43

44

7.6.1 Neighborhood Gather

⁴⁵ In this function, each process i gathers data items from each process j if an edge (j, i) exists ⁴⁶ in the topology graph, and each process i sends the same data items to all processes j where ⁴⁷ an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block ⁴⁸ in the receive buffer is received from the *l*-th neighbor.

```
MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                           1
                                                                                          \mathbf{2}
               comm)
                                                                                          3
  IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                          4
  IN
           sendcount
                                       number of elements sent to each neighbor (non-negative
                                                                                          5
                                       integer)
                                                                                          6
                                                                                          7
  IN
           sendtype
                                       data type of send buffer elements (handle)
                                                                                           8
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                          9
                                       number of elements received from each neighbor (non-
  IN
           recvcount
                                                                                          10
                                       negative integer)
                                                                                          11
  IN
           recvtype
                                       data type of receive buffer elements (handle)
                                                                                          12
                                                                                          13
                                       communicator with topology structure (handle)
  IN
           comm
                                                                                          14
                                                                                          15
                                                                                          _{16} ticket 140.
int MPI_Neighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype
               sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
                                                                                          17
               MPI_Comm comm)
                                                                                          18
                                                                                            ticket-248T.
                                                                                          19
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                          20
               recvtype, comm, ierror) BIND(C)
                                                                                          21
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                          22
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                          23
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                          ^{24}
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          27
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                          28
               RECVTYPE, COMM, IERROR)
                                                                                          29
    <type> SENDBUF(*), RECVBUF(*)
                                                                                          30
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
                                                                                          ^{31}
                                                                                          32
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                          33
graph communicators as described in Section 7.6 on page 328. If comm is a distributed graph
                                                                                          34
communicator, the outcome is as if each process executed sends to each of its outgoing
                                                                                          35
neighbors and receives from each of its incoming neighbors:
                                                                                          36
                                                                                          37
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                          38
                                                                                          39
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                          40
                                                                                          41
                            outdegree,dsts,MPI_UNWEIGHTED);
                                                                                          42
int k,l;
                                                                                          43
                                                                                          44
for(k=0; k<outdegree; ++k)</pre>
  MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
                                                                                          45
                                                                                          46
                                                                                          47
for(1=0; l<indegree; ++1)</pre>
                                                                                          48
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
```

1	<pre>srcs[1],);</pre>
3	<pre>MPI_Waitall();</pre>
4	
5 6 7 8	Figure 7.6.1 shows the neighborhood gather communication of one process with out- going neighbors $d_0 \ldots d_3$ and incoming neighbors $s_0 \ldots s_5$. The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.
9	d
10	
11 12	d_2, s_4
13	s_0
14	d_1 s_1
15	
16	
17	s_2 S_3
18	d_3, s_5
19 20	sendbuf
20	
22	
23	s_0 s_1 s_2 s_3 s_4 s_5
24	recvbuf
25	
26	All arguments are significant on all processes and the argument
27	comm must have identical values on all processes.
28 29	The type signature associated with sendcount, sendtype, at a process must be equal to
30	the type signature associated with recvcount , recvtype at all other processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between
31	every pair of communicating processes. Distinct type maps between sender and receiver are
32	still allowed.
33	
34	Rationale. For optimization reasons, the same type signature is required indepen-
35	dently of whether the topology graph is connected or not. (<i>End of rationale.</i>)
36	
37	The "in place" option is not meaningful for this operation. The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different
38 39	numbers of elements from each neighbor.
40	numbers of elements from each heighbor.
41	
42	
43	
44	
45	
46	
47	
48	

```
MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                             1
                                                                                             \mathbf{2}
               recvtype, comm)
                                                                                             3
  IN
           sendbuf
                                        starting address of send buffer (choice)
                                                                                            4
  IN
            sendcount
                                         number of elements sent to each neighbor (non-negative
                                                                                             5
                                        integer)
                                                                                             6
                                                                                             7
  IN
           sendtype
                                        data type of send buffer elements (handle)
                                                                                             8
  OUT
           recvbuf
                                        starting address of receive buffer (choice)
                                                                                             9
                                        non-negative integer array (of length indegree) con-
  IN
            recvcounts
                                                                                            10
                                        taining the number of elements that are received from
                                                                                            11
                                         each neighbor
                                                                                            12
  IN
                                        integer array (of length indegree). Entry i specifies
                                                                                            13
            displs
                                                                                            14
                                        the displacement (relative to recvbuf) at which to place
                                                                                            15
                                         the incoming data from neighbor i
                                                                                            16
  IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                            17
  IN
                                        communicator with topology structure (handle)
           comm
                                                                                            18
                                                                                            19
                                                                                            <sup>20</sup> ticket140.
int MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount,
               MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
                                                                                            ^{21} ticket 140.
                                                                                            <sup>22</sup> ticket140.
               const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
                                                                                            <sup>23</sup> ticket-248T.
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                            24
               displs, recvtype, comm, ierror) BIND(C)
                                                                                            25
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                            26
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                            27
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                            28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                            29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                            30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            31
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
                                                                                            32
               DISPLS, RECVTYPE, COMM, IERROR)
                                                                                            33
    <type> SENDBUF(*), RECVBUF(*)
                                                                                            34
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
                                                                                            35
    IERROR
                                                                                            36
                                                                                            37
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                            38
graph communicators as described in Section 7.6 on page 328. If comm is a distributed graph
                                                                                            39
communicator, the outcome is as if each process executed sends to each of its outgoing
                                                                                            40
neighbors and receives from each of its incoming neighbors:
                                                                                            41
                                                                                            42
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
                                                                                            43
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                            44
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                            45
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                            46
                             outdegree,dsts,MPI_UNWEIGHTED);
                                                                                            47
```

int k,l;

```
1
             \mathbf{2}
                  for(k=0; k<outdegree; ++k)</pre>
             3
                     MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
             4
             \mathbf{5}
                  for(l=0; l<indegree; ++1)</pre>
             6
                    MPI_Irecv(recvbuf+displs[l]*extent(recvtype),recvcounts[l],recvtype,
             7
                                 srcs[1],...);
             8
             9
                  MPI_Waitall(...);
            10
                       The type signature associated with sendcount, sendtype, at process j must be equal
            11
                  to the type signature associated with recvcounts[1], recvtype at any other process with
            12
                  srcs[1]==j. This implies that the amount of data sent must be equal to the amount of
            13
                  data received, pairwise between every pair of communicating processes. Distinct type maps
            14
                  between sender and receiver are still allowed. The data received from the 1-th neighbor is
            15
            16
                  placed into recvbuf beginning at offset displs[1] elements (in terms of the recvtype).
                       The "in place" option is not meaningful for this operation.
            17
                       All arguments are significant on all processes and the argument
            18
                  comm must have identical values on all processes.
            19
            20
            21
                  7.6.2
                          Neighbor Alltoall
            22
                  In this function, each process i receives data items from each process j if an edge (j,i)
            23
                  exists in the topology graph or Cartesian topology. Similarly, each process i sends data
            24
                  items to all processes j where an edge (i, j) exists. This call is more general than
            25
                  MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor.
            26
                  The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in
            27
                  the receive buffer is received from the l-th neighbor.
            28
            29
            30
                  MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
            ^{31}
            32
                    IN
                               sendbuf
                                                             starting address of send buffer (choice)
            33
            34
                    IN
                               sendcount
                                                             number of elements sent to each neighbor (non-negative
            35
                                                             integer)
            36
                    IN
                               sendtype
                                                             data type of send buffer elements (handle)
            37
                    OUT
                               recvbuf
                                                             starting address of receive buffer (choice)
            38
            39
                    IN
                                                             number of elements received from each neighbor (non-
                               recvcount
            40
                                                             negative integer)
            41
                    IN
                                                             data type of receive buffer elements (handle)
                               recvtype
            42
                    IN
                                                             communicator with topology structure (handle)
                               comm
            43
            44
                  int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype
  ticket140. 45
                                  sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
            46
                                  MPI_Comm comm)
ticket-248T. \frac{1}{48}
```

```
1
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                         2
               recvtype, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                         3
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                         4
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                         5
                                                                                         6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         9
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                         10
               RECVTYPE, COMM, IERROR)
                                                                                         11
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         12
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
                                                                                         13
                                                                                        14
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                         15
graph communicators as described in Section 7.6 on page 328. If comm is a distributed graph
                                                                                         16
communicator, the outcome is as if each process executed sends to each of its outgoing
                                                                                         17
neighbors and receives from each of its incoming neighbors:
                                                                                        18
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
                                                                                        19
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                        20
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                        21
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                        22
                            outdegree,dsts,MPI_UNWEIGHTED);
                                                                                        23
                                                                                         24
int k,l;
                                                                                         25
                                                                                         26
for(k=0; k<outdegree; ++k)</pre>
  MPI_Isend(sendbuf+k*sendcount*extent(sendtype),sendcount,sendtype,
                                                                                        27
             dsts[k],...);
                                                                                        28
                                                                                        29
for(l=0; l<indegree; ++1)</pre>
                                                                                         30
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
                                                                                         31
             srcs[1],...);
                                                                                         32
                                                                                         33
MPI_Waitall(...);
                                                                                        34
                                                                                        35
    The type signature associated with sendcount, sendtype, at a process must be equal to
                                                                                        36
the type signature associated with recvcount, recvtype at any other process. This implies
                                                                                        37
that the amount of data sent must be equal to the amount of data received, pairwise between
                                                                                        38
every pair of communicating processes. Distinct type maps between sender and receiver are
                                                                                        39
still allowed.
                                                                                         40
    The "in place" option is not meaningful for this operation.
                                                                                         41
    All arguments are significant on all processes and the argument
                                                                                         42
comm must have identical values on all processes.
                                                                                        43
    The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
                                                                                        44
numbers of elements to and from each neighbor.
                                                                                         45
                                                                                         46
                                                                                         47
```

1 2	MPI_NEI	GHBOR_ALLTOALL rdispls, recvty	/(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, pe, comm)
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
7 8 9	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which to send the outgoing data to neighbor j
10	IN	sendtype	data type of send buffer elements (handle)
11 12	OUT	recvbuf	starting address of receive buffer (choice)
13 14 15	IN	recvcounts	non-negative integer array (of length indegree) spec- ifying the number of elements that can are received from each neighbor
16 17 18 19	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
20	IN	recvtype	data type of receive buffer elements (handle)
21	IN	comm	communicator with topology structure (handle)
<pre>ticket140.²³ ticket140.²⁴ ticket140.²⁵ ticket140.²⁶ ticket140.²⁷ ticket140.²⁷ ticket248T.²⁸ MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, ²⁹ ³⁰ TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf ³¹ TYPE(*), DIMENSION(), intENT(IN) :: sendbuf ³² INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcount ³³ rdispls(*) ³⁴ TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype ³⁵ TYPE(MPI_Comm), INTENT(IN) :: ierror ³⁶ MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, ³⁸ RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) ³⁹ <type> SENDBUF(*), RECVBUF(*) ³⁰ TYPE(*), COMM, IERROR ³¹ TYPE, COMM, IERROR ³² This function supports Cartesian communicators, graph communicators ³⁴ graph communicators as described in Section 7.6 on page 328. If comm is a d</type></pre>		displs[], MPI_Datatype sendtype, void* recvbuf, ecvcounts[], const int rdispls[], MPI_Datatype	
		recvcounts, (*), DIMENSION(. (*), DIMENSION(. GER, INTENT(IN) pls(*) (MPI_Datatype), (MPI_Comm), INTE GER, OPTIONAL, I HBOR_ALLTOALLV(S RECVCOUNTS, pe> SENDBUF(*), I GER SENDCOUNTS(* TYPE, COMM, IERR function supports C amunicators as description	<pre>rdispls, recvtype, comm, ierror) BIND(C) .), INTENT(IN) :: sendbuf .) :: recvbuf :: sendcounts(*), sdispls(*), recvcounts(*), INTENT(IN) :: sendtype, recvtype VT(IN) :: comm VTENT(OUT) :: ierror ENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RDISPLS, RECVTYPE, COMM, IERROR) RECVBUF(*)), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), OR artesian communicators, graph communicators, and distributed ribed in Section 7.6 on page 328. If comm is a distributed graph</pre>
46 47 48			each of its incoming neighbors:

```
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                               \mathbf{2}
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                               3
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                               4
                              outdegree,dsts,MPI_UNWEIGHTED);
int k,l;
                                                                                               5
                                                                                               6
                                                                                               7
for(k=0; k<outdegree; ++k)</pre>
  MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
              dsts[k],...);
                                                                                               9
                                                                                              10
                                                                                              11
for(1=0; l<indegree; ++1)</pre>
  MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype), recvcounts[1], recvtype,
                                                                                              12
              srcs[1],...);
                                                                                              13
                                                                                              14
                                                                                              15
MPI_Waitall(...);
                                                                                              16
    The type signature associated with sendcounts[k], sendtype with dsts[k] == j at pro-
                                                                                              17
cess i must be equal to the type signature associated with recvcounts[1], recvtype with
                                                                                              18
srcs[1]==i at process j. This implies that the amount of data sent must be equal to the
                                                                                              19
amount of data received, pairwise between every pair of communicating processes. Distinct
                                                                                              20
type maps between sender and receiver are still allowed. The data in the sendbuf beginning
                                                                                              21
at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor.
                                                                                              22
The data received from the 1-th incoming neighbor is placed into recvbuf beginning at offset
                                                                                              23
                                                                                              ^{24}
rdispls[1] elements (in terms of the recvtype).
    The "in place" option is not meaningful for this operation.
                                                                                              25
    All arguments are significant on all processes and the argument
                                                                                              26
comm must have identical values on all processes.
                                                                                              27
     MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes
                                                                                              28
to and from each neighbor.
                                                                                              29
                                                                                              30
                                                                                              31
                                                                                              32
                                                                                              33
                                                                                              34
                                                                                              35
                                                                                              36
                                                                                              37
                                                                                              38
                                                                                              39
                                                                                              40
                                                                                              41
                                                                                              42
                                                                                              43
                                                                                              44
                                                                                              45
                                                                                              46
                                                                                              47
                                                                                              48
```

1 2	MPI_NEI		LW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, types, comm)
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
7	IN	sdispls	integer array (of length outdegree). Entry j specifies
8	IIN	3013013	the displacement in bytes (relative to sendbuf) from
9			which to take the outgoing data destined for neighbor
10			j (array of integers)
11	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec-
12			ifies the type of data to send to neighbor j (array of
13			handles)
14 15	OUT	recvbuf	starting address of receive buffer (choice)
16	IN	recvcounts	non-negative integer array (of length indegree) spec-
17			ifying the number of elements that can are received
18			from each neighbor
19	IN	rdispls	integer array (of length indegree). Entry i specifies
20			the displacement in bytes (relative to $recvbuf$) at which
21 22			to place the incoming data from neighbor i (array of
22			integers)
24	IN	recvtypes	array of datatypes (of length indegree). Entry i spec-
25 26			ifies the type of data received from neighbor i (array of handles)
27	IN	comm	communicator with topology structure (handle)
<pre>ticket140.31 ticket299.32 ticket140.33 ticket140.34 ticket140.35 ticket140.36 ticket140.37 ticket-248T.37 </pre>		<pre>[]MPI_Aint sdispls[], const MPI_Datatype sendtypes[], rbuf, const int recvcounts[], const [int]MPI_Aint const MPI_Datatype recvtypes[], MPI_Comm comm) sendbuf, sendcounts, sdispls, sendtypes, recvbuf, s, rdispls, recvtypes, comm, ierror) BIND(C)), INTENT(IN) :: sendbuf) :: recvbuf :: sendcounts(*), recvcounts(*) DRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*) INTENT(IN) :: sendtypes(*), recvtypes(*) ENT(IN) :: comm INTENT(OUT) :: ierror SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, S, RDISPLS, RECVTYPES, COMM, IERROR) RECVBUF(*)</pre>	
47 48	INTE IERF		<pre>*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,</pre>
	TENL	0016	

 $\mathbf{2}$

4

5

6

7

8

9

10

11 12

13

1415

16

17 18

19 20

21

22

23

 24

25

26

27

28 29 30

 31 32

33

34

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 328. If comm is a distributed graph 3 communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                         outdegree,dsts,MPI_UNWEIGHTED);
```

int k,l;

```
for(k=0; k<outdegree; ++k)</pre>
  MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);
for(1=0; l<indegree; ++1)</pre>
```

```
MPI_Irecv(recvbuf+rdispls[1], recvcounts[1], recvtypes[1], srcs[1], ...);
```

MPI_Waitall(...);

The type signature associated with sendcounts [k], sendtypes [k] with dsts [k] == j at process i must be equal to the type signature associated with recvcounts[1], recvtypes[1] with srcs[1]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

Nonblocking Neighborhood Communication on Process Topologies 7.7

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 5.12.

	338		CHAPTER 7. PROCESS TOPOLOGIES
1	7.7.1	Nonblocking Neighbo	orhood Gather
2			
4			ER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
5		comm, reque	
6	IN	sendbuf	starting address of send buffer (choice)
7	IN	sendcount	number of elements sent to each neighbor (non-negative
9	IN	senacount	integer)
10 11	IN	sendtype	data type of send buffer elements (handle)
11	OUT	recvbuf	starting address of receive buffer (choice)
13 14	IN	recvcount	number of elements received from each neighbor (non-negative integer)
15	IN	recvtype	data type of receive buffer elements (handle)
16 17	IN	comm	communicator with topology structure (handle)
18	OUT	request	communication request (handle)
19			
ticket140. 20 21 22 ticket-248T. 23 24 25 26 27 28 29 30 31	<pre>int MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,</pre>		
32	INT	EGER, OPTIONAL, I	NTENT(OUT) :: ierror
33 34	MPI_INE		SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR)
35	< t.v	vpe> SENDBUF(*),	
36	•		SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37	Thi	e call starte a nonble	ocking variant of MPI_NEIGHBOR_ALLGATHER.
38 39	1 111	s can starts a nondic	CKIIIg Variant of WHILMEIGHDON_ALEGATHEN.
40			
41			
42			
43			
44			
45			
46			
47 48			
40			

MPI_	NEIGHBOR_ALLGATHERV recvtype, comm, r	(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, request)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
			4
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	5
INI	and the set		6 7
IN	sendtype	data type of send buffer elements (handle)	8
00-	Г recvbuf	starting address of receive buffer (choice)	9
IN	recvcounts	non-negative integer array (of length indegree) con-	10
		taining the number of elements that are received from	11
		each neighbor	12
IN	displs	integer array (of length indegree). Entry i specifies	13
		the displacement (relative to recvbuf) at which to place	14 15
		the incoming data from neighbor i	16
IN	recvtype	data type of receive buffer elements (handle)	17
IN	comm	communicator with topology structure (handle)	18
00	Г request	communication request (handle)	19
			20
int M	PI_Ineighbor_allgatherv	(const void* sendbuf, int sendcount,	21 ticket 140.
	• •	<pre>endtype, void* recvbuf, const int recvcounts[],</pre>	$^{22}_{23}$ ticket 140.
		<pre>ls[], MPI_Datatype recvtype, MPI_Comm comm, ```````````````````````````````````</pre>	$^{23}_{24}$ ticket 140.
	MPI_Request *r	equest)	25 ticket-248T.
MPI_1	e	ndbuf, sendcount, sendtype, recvbuf, recvcounts,	26
		pe, comm, request, ierror) BIND(C)	27
		INTENT(IN), ASYNCHRONOUS :: sendbuf	28
	<pre>YPE(*), DIMENSION(), INTEGER, INTENT(IN) ::</pre>	ASYNCHRONOUS :: recvbuf	29
	-	<pre>XNCHRONOUS :: recvcounts(*), displs(*)</pre>	30 31
		ENT(IN) :: sendtype, recvtype	32
	YPE(MPI_Comm), INTENT(]		33
1	YPE(MPI_Request), INTEN	NT(OUT) :: request	34
]	NTEGER, OPTIONAL, INTEN	NT(OUT) :: ierror	35
MPI_I	NEIGHBOR_ALLGATHERV(SEN	IDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	36
		PE, COMM, REQUEST, IERROR)	37
	<type> SENDBUF(*), RECV</type>		38 39
		TYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	40
F	EQUEST, IERROR		41
ſ	This call starts a nonblockin	g variant of MPI_NEIGHBOR_ALLGATHERV.	42
			43
			44
			45

	340		CHAPTER 7. PROCESS TOPOLOGIES	
1 2 3	7.7.2 N	lonblocking Neighbo	ing Neighborhood Alltoall	
4 5	MPI_INE	IGHBOR_ALLTOALL request)	(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,	
6 7	IN	sendbuf	starting address of send buffer (choice)	
8 9	IN	sendcount	number of elements sent to each neighbor (non-negative integer)	
10	IN	sendtype	data type of send buffer elements (handle)	
11	OUT	recvbuf	starting address of receive buffer (choice)	
12 13 14	IN	recvcount	number of elements received from each neighbor (non- negative integer)	
15	IN	recvtype	data type of receive buffer elements (handle)	
16 17	IN	comm	communicator with topology structure (handle)	
18	OUT	request	communication request (handle)	
19				
ticket140. 20 21 22 ticket-248T. 23 24 25 26 27 28 29 30 31 32 33 34 35	MPI_Ine: TYPI TYPI INTI TYPI TYPI INTI MPI_INE:	<pre>MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount, MPI_Data sendtype, void* recvbuf, int recvcount, MPI_Datatype recvt MPI_Comm comm, MPI_Request *request) .Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount recvtype, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror .INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT RECVTYPE, COMM, REQUEST, IERROR)</pre>		
36 37 38 39 40 41 42 43 44 45 46 47 48	INTI		RECVBUF(*) ENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR eking variant of MPI_NEIGHBOR_ALLTOALL.	

MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, 1 $\mathbf{2}$ rdispls, recvtype, comm, request) 3 IN sendbuf starting address of send buffer (choice) 4 IN sendcounts non-negative integer array (of length outdegree) speci-5fying the number of elements to send to each neighbor 6 7IN sdispls integer array (of length outdegree). Entry j specifies 8 the displacement (relative to sendbuf) from which send 9 the outgoing data to neighbor j 10 IN sendtype data type of send buffer elements (handle) 11 OUT recvbuf starting address of receive buffer (choice) 12IN non-negative integer array (of length indegree) spec-13 recvcounts 14ifying the number of elements that can are received 15from each neighbor 16IN rdispls integer array (of length indegree). Entry i specifies 17 the displacement (relative to recvbuf) at which to place 18 the incoming data from neighbor i 19IN data type of receive buffer elements (handle) recvtype 2021IN communicator with topology structure (handle) comm 22 OUT communication request (handle) request 2324int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[], ₂₅ ticket140. $_{26}$ ticket 140. const int sdispls[], MPI_Datatype sendtype, void* recvbuf, ticket140. const int recvcounts[], const int rdispls[], MPI_Datatype 27ticket140. recvtype, MPI_Comm comm, MPI_Request *request) 28ticket140. 29 ticket-248T. MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, 30 recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C) 31 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 32 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 33 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), 34 recvcounts(*), rdispls(*) 35TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 36 TYPE(MPI_Comm), INTENT(IN) :: comm 37 TYPE(MPI_Request), INTENT(OUT) :: request 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 40 RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 41 <type> SENDBUF(*), RECVBUF(*) 42INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 43 RECVTYPE, COMM, REQUEST, IERROR 4445This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV. 4647

	1 2	MPI_INEI	GHBOR_ALLTOALLW(sendbornd) rdispls, recvtypes, com	uf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, m, request)
	3	IN	sendbuf	starting address of send buffer (choice)
	4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
	7 8 9 10	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
	11 12 13 14	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)
	15	OUT	recvbuf	starting address of receive buffer (choice)
	16 17 18	IN	recvcounts	non-negative integer array (of length indegree) spec- ifying the number of elements that can are received from each neighbor
	19 20 21 22 23	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
	24 25 26	IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)
	27	IN	comm	communicator with topology structure (handle)
	28 29	OUT	request	communication request (handle)
ticket140. ticket140. ticket140. ticket299. ticket140. ticket140.	32 33 34 35	int MPI_]	<pre>const [int]MPI_Aint void* recvbuf, cons</pre>	st void* sendbuf, const int sendcounts[], t sdispls[], const MPI_Datatype sendtypes[], st int recvcounts[], const [int]MPI_Aint PI_Datatype recvtypes[], MPI_Comm comm, st)
ticket140. ticket299. ticket140. ticket-248T.	36 37 38	TYPE(TYPE) INTE(INTE(sdisp TYPE) recvt TYPE(TYPE)	recvcounts, rdispla (*), DIMENSION(), INTE (*), DIMENSION(), ASYN GER, INTENT(IN), ASYNCHE GER(KIND=MPI_ADDRESS_KIN pls(*), rdispls(*)	<pre>RONOUS :: sendcounts(*), recvcounts(*) ND), INTENT(IN), ASYNCHRONOUS :: IN), ASYNCHRONOUS :: sendtypes(*), :: comm JT) :: request</pre>

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.

7.8 An Application Example

Example 7.9 []The example in [Figure 7.1]Figures 7.2-7.4 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine relax.

In each relaxation step each process computes new values for the solution grid function at [all]the points u(1:100,1:100) owned by the process. Then the values at interprocess boundaries have to be exchanged with neighboring processes. For example, the [exchange subroutine might contain a call like MPI_SEND(...,neigh_rank(1),...) to send updated values to the left-hand neighbor (i-1,j).]newly calculated values in u(1,1:100)must be sent into the halo cells u(101,1:100) of the left-hand neighbor with coordinates (own_coord(1)-1,own_coord(2))

56 7 8 9 10 11 12 $_{13}$ ticket 258. $_{14}$ ticket 258. $_{15}$ ticket258. ticket258. 16 17 18 19 20 $_{21}$ ticket 258. $_{22}$ ticket 258. 23 ticket 258. 24 25 $_{26}$ ticket 258. $_{27}$ ticket258. $_{28}$ ticket258. $_{29}$ ticket258. 30 31 32 33 34 3536 37 38 39 40 41 4243 4445464748

1

 $\mathbf{2}$

3

```
2
          integer ndims, num_neigh
3
          logical reorder
4
          parameter (ndims=2, num_neigh=4, reorder=.true.)
5
          integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
6
          integer neigh_rank(num_neigh), own_position(ndims), i, j
7
          logical periods(ndims)
8
          real*8 u(0:101,0:101), f(0:101,0:101)
9
          data dims / ndims * 0 /
10
          comm = MPI_COMM_WORLD
11
     С
           Set process grid size and periodicity
12
          call MPI_DIMS_CREATE(comm, ndims, dims, ierr)
13
          periods(1) = .TRUE.
14
          periods(2) = .TRUE.
15
     С
          Create a grid structure in WORLD group and inquire about own position
16
          call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
17
          call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
18
           Look up the ranks for the neighbors. Own process coordinates are (i,j).
     С
19
          Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
     С
20
          i = own_position(1)
21
          j = own_position(2)
22
          neigh_def(1) = i-1
23
          neigh_def(2) = j
24
          call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
25
          neigh_def(1) = i+1
26
          neigh_def(2) = j
27
          call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
28
          neigh_def(1) = i
29
          neigh_def(2) = j-1
30
          call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
31
          neigh_def(1) = i
32
          neigh_def(2) = j+1
33
          call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
34
     С
           Initialize the grid functions and start the iteration
35
          call init (u, f)
36
          do 10 it=1,100
37
            call relax (u, f)
38
     С
           Exchange data with neighbor processes
39
            call exchange (u, comm_cart, neigh_rank, num_neigh)
40
          continue
     10
41
          call output (u)
42
          end
43
44
45
46
        Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.
47
48
```

```
6
                                                                                     7
INTEGER ndims, num_neigh
                                                                                     8
LOGICAL reorder
                                                                                     9
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                     10
                                                                                     11
INTEGER comm, comm_cart, dims(ndims), ierr
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
                                                                                    12
LOGICAL periods(ndims)
                                                                                    13
REAL u(0:101,0:101), f(0:101,0:101)
                                                                                    14
DATA dims / ndims * 0 /
                                                                                     15
comm = MPI_COMM_WORLD
                                                                                     16
                                                                                     17
!
    Set process grid size and periodicity
                                                                                     18
CALL MPI_DIMS_CREATE(comm, ndims, dims,ierr)
periods(1) = .TRUE.
                                                                                     19
periods(2) = .TRUE.
                                                                                    20
    Create a grid structure in WORLD group and inquire about own position
                                                                                    21
CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
                                                                                    22
                   comm_cart,ierr)
                                                                                    23
                                                                                    ^{24}
CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords,ierr)
                                                                                    25
i = own_coords(1)
                                                                                    26
j = own_coords(2)
    Look up the ranks for the neighbors. Own process coordinates are (i,j).
                                                                                    27
Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
                                                                                    28
CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
                                                                                    29
CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3),neigh_rank(4), ierr)
                                                                                    30
    Initialize the grid functions and start the iteration
                                                                                     31
!
CALL init (u, f)
                                                                                     32
                                                                                     33
DO it=1,100
                                                                                    34
   CALL relax (u, f)
!
       Exchange data with neighbor processes
                                                                                    35
   CALL exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                    36
END DO
                                                                                    37
CALL output (u)
                                                                                    38
                                                                                    39
                                                                                     40
   Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                     44
```

```
1
\mathbf{2}
3
4
5
6
7
8
9
10
11
     SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
12
     REAL u(0:101,0:101)
13
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
14
     REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
15
     INTEGER ierr
16
     sndbuf(1:100,1) = u( 1,1:100)
17
     sndbuf(1:100,2) = u(100,1:100)
18
     sndbuf(1:100,3) = u(1:100, 1)
19
     sndbuf(1:100,4) = u(1:100,100)
20
     CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
21
                                  comm_cart, ierr)
22
     ! instead of
23
     ! DO i=1,num_neigh
24
         CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),ierr)
     !
25
         CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),ierr)
     !
26
     ! END DO
27
     ! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
28
29
     u( 0,1:100) = rcvbuf(1:100,1)
30
     u(101,1:100) = rcvbuf(1:100,2)
31
     u(1:100, 0) = rcvbuf(1:100,3)
32
     u(1:100,101) = rcvbuf(1:100,4)
33
     END
34
35
36
     Figure 7.3: Communication routine with local data copying and sparse neighborhood all-
37
     to-all.
38
39
40
41
42
43
44
45
46
47
48
```

```
2
                                                                                  3
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                  4
USE MPI
                                                                                  5
REAL u(0:101,0:101)
                                                                                  6
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                  7
INTEGER sndcounts(num_neigh), sdispls(num_neigh), sndtypes(num_neigh)
                                                                                  8
INTEGER rcvcounts(num_neigh), rdispls(num_neigh), rcvtypes(num_neigh)
                                                                                  9
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
                                                                                  10
INTEGER type_vec, i, ierr
                                                                                  11
!
    The following initialization need to be done only once
                                                                                  12
!
    before the first call of exchange.
                                                                                  13
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
                                                                                  14
CALL MPI_TYPE_VECTOR (100, 1, 102, MPI_REAL, type_vec, ierr)
                                                                                  15
CALL MPI_TYPE_COMMIT (type_vec, ierr)
                                                                                  16
sndtypes(1) = type_vec
                                                                                  17
sndtypes(2) = type_vec
                                                                                  18
sndtypes(3) = MPI_REAL
                                                                                  19
sndtypes(4) = MPI_REAL
                                                                                  20
DO i=1,num_neigh
                                                                                  21
   sndcounts(i) = 100
                                                                                  22
   rcvcounts(i) = 100
                                                                                  23
   rcvtypes(i) = sndtypes(i)
                                                                                  24
END DO
                                                                                  25
sdispls(1) = (1 +
                      1*102) * sizeofreal
                                              ! first element of u( 1,1:100)
                                                                                  26
sdispls(2) = (100 +
                     1*102) * sizeofreal ! first element of u(100,1:100)
                                                                                  27
sdispls(3) = (1 + 1*102) * size of real ! first element of u(1:100, )
                                                                            1)
                                                                                  28
sdispls(4) = ( 1 + 100*102) * sizeofreal
                                             ! first element of u(1:100,100)
                                                                                  29
rdispls(1) = (0 +
                      1*102) * sizeofreal
                                             ! first element of u( 0,1:100)
                                                                                  30
rdispls(2) = (101 +
                      1*102) * sizeofreal ! first element of u(101,1:100)
                                                                                  31
rdispls(3) = ( 1 +
                      0*102) * sizeofreal ! first element of u(1:100, 0)
                                                                                  32
rdispls(4) = ( 1 + 101*102) * sizeofreal ! first element of u(1:100,101)
                                                                                  33
                                                                                  34
! the following communication has to be done in each call of exchange
                                                                                  35
CALL MPI_NEIGHBOR_ALLTOALLW (u, sndcounts, sdispls, sndtypes, &
                                                                                  36
                            u, rcvcounts, rdispls, rcvtypes, comm_cart, ierr)
                                                                                  37
                                                                                  38
1
    The following finalizing need to be done only once
                                                                                  39
    after the last call of exchange.
40
CALL MPI_TYPE_FREE (type_vec, ierr)
                                                                                  41
END
                                                                                  42
                                                                                  43
                                                                                  44
Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local
                                                                                  45
data copying.
                                                                                  46
```

```
Unofficial Draft for Comment Only
```

Chapter 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C and C++, [

```
#define MPI_VERSION 2
#define MPI_SUBVERSION 2
```

in Fortran, HEADER SKIP ENDHEADER

```
INTEGER [ticket240-L.]:: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION = 2)
PARAMETER (MPI_SUBVERSION = 2)
```


#define MPI_VERSION 3
#define MPI_SUBVERSION 0

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION = 3)
PARAMETER (MPI_SUBVERSION = 0)
```

For runtime determination,

²⁷ ticket0-unch

 $46 \\ 47$

```
1
                  MPI_GET_VERSION( version, subversion )
            2
                    OUT
                              version
                                                           version number (integer)
             3
                    OUT
                              subversion
                                                           subversion number (integer)
             4
            5
             6
                  int MPI_Get_version(int *version, int *subversion)
ticket-248T.
                  MPI_Get_version(version, subversion, ierror) BIND(C)
             8
                       INTEGER, INTENT(OUT) :: version, subversion
            9
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            10
            11
                  MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
            12
                       INTEGER VERSION, SUBVERSION, IERROR
            13
                  {void MPI::Get_version(int& version, int& subversion)(binding deprecated, see
            14
                                  Section 15.2 }
            15
  ticket204.
            16
            17
                       MPI_GET_VERSION is one of the few functions that can be called ]
            18
                      before MPI_INIT and after MPI_FINALIZE. Valid (MPI_VERSION, MPI_SUBVERSION)
0-unchecked.<sup>19</sup>
                  pairs in this and previous versions of the MPI standard are [(3,0), (2,2), (2,1), (2,0), (2,0)]
  ticket204. 20
                  (1,2).
            21
            22
                  MPI_GET_LIBRARY_VERSION( version, resultlen )
            23
            ^{24}
                    OUT
                              version
                                                           version string (string)
            25
                    OUT
                              resultlen
                                                           Length (in printable characters) of the result returned
            26
                                                           in version (integer)
            27
            28
                  int MPI_Get_library_version(char *version, int *resultlen)
            29
            30
                  MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR)
            ^{31}
                       CHARACTER*(*) VERSION
            32
                       INTEGER RESULTLEN, IERROR
            33
                      This routine returns a string representing the version of the MPI library. The version
            34
                  argument is a character string for maximum flexibility.
            35
            36
                        Advice to implementors. An implementation of MPI should return a different string
            37
                       for every change to its source code or build that could be visible to the user. (End of
            38
                        advice to implementors.)
            39
            40
                      The argument version must represent storage that is MPI_MAX_LIBRARY_VERSION_STRING
            41
                  characters long. MPI_GET_LIBRARY_VERSION may write up to this many characters into
            42
                  version.
            43
                      The number of characters actually written is returned in the output argument, resultlen.
            44
  ticket207.
                  In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot
            45
                  be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on
  ticket207. _{47}
                  the right with blank characters. The value of resultlen cannot be larger than
            48
```

MPI_MAX_LIBRARY_VERSION_STRING.	1				
$MPI_GET_VERSION$ and $MPI_GET_LIBRARY_VERSION$ are two of the few functions					
that can be called before MPI_INIT and after MPI_FINALIZE.					
	4				
8.1.2 Environmental Inquiries	5				
A set of attributes that describe the execution environment are attached to the commu-	6				
nicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can	7 8				
be inquired by using the function MPI_COMM_GET_ATTR described in Chapter 6. It is	8 9				
erroneous to delete these attributes, free their keys, or change their values.	10				
The list of predefined attribute keys include					
* *	11 12				
MPI_TAG_UB Upper bound for tag value.	13				
MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.	14				
WFT_HOST HOSt process rank, if such exists, WFT_FROC_NOLL, otherwise.	15				
MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same	16				
communicator may return different values for this parameter.	17				
	18				
MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.	19				
Vendors may add implementation specific parameters (such as node number, real mem-	20				
ory size, virtual memory size, etc.)	21				
These predefined attributes do not change value between MPI initialization (MPI_INIT)	22 ticket 215.				
and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	23				
Advice to users. Note that in the C binding, the value returned by these attributes	25				
is a <i>pointer</i> to an <i>int</i> containing the requested value. (End of advice to users.)					
	27				
The required parameter values are discussed in more detail below:	28				
	29				
Tag Values	30				
Tag values range from 0 to the value returned for MPI_TAG_UB inclusive. These values are	31				
guaranteed to be unchanging during the execution of an MPI program. In addition, the tag	32				
upper bound value must be at least 32767. An MPI implementation is free to make the	33				
value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a [legal] valid	³⁴				
value of MPI_TAG_UB larger than this, for example, the value 2^{-} – 1 is also a [legal]value value for MPI_TAG_UB.	$^{34}_{35}$ ticket182.				
The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.	36				
The attribute MFT_TAG_OD has the same value on an processes of MFT_COMM_WORLD.	37				
Heat Deals	38				
Host Rank	39				
The value returned for MPI_HOST gets the rank of the HOST process in the group associated	40				
with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if	41				
there is no host. MPI does not specify what it means for a process to be a HOST, nor does	42				
it requires that a HOST exists.	43				
The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.					
	48				

	1	IO Rank				
	2 3		_IO is the rank of a processor that can provide language-standard			
	4	this means that all of the Fortran I/O operations are supported). For C and C++, this means that all of the ISO C and C++,				
	5 6	I/O operations are supported (e.g., fopen, fprintf, lseek).If every process can provide language-standard I/O, then the value MPI_ANY_will be returned. Otherwise, if the calling process can provide language-standard				
	7					
	8					
	9	$_{9}$ then its rank will be returned. Otherwise, if some process can provide language I/O then the rank of one such process will be returned. The same value ne				
	10	1	such process will be returned. The same value need not be if no process can provide language-standard I/O , then the value			
	11 12	MPI_PROC_NULL will be re	, .			
	13	Aduites to second Net				
	14		the that input is not collective, and this attribute does <i>not</i> indicate does provide input. (<i>End of advice to users.</i>)			
	15	which process can of	does provide input. (<i>Linu of unoice to users.</i>)			
	16 17	Clock Synchronization				
	18	The value returned for	MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in			
	19	MPI_COMM_WORLD are sy	rnchronized, 0 otherwise. A collection of clocks is considered			
	20	· ·	ort has been taken to synchronize them. The expectation is that			
	21 22		heasured by calls to MPI_WTIME, will be less then one half the			
	23		I message of length zero. If time is measured at a process just or process just after a matching receive, the second time should			
	24	before a send and at another process just after a matching receive, the second time should be always higher than the first one.				
	25	The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are no synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This				
	26					
	27	The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of				
	28 29					
	²⁹ MPI_COMM_WORLD. ³⁰					
ticket 255.	31 32	Inquire Processor Name				
	33					
	34	MPI_GET_PROCESSOR_N	AME(name resultien)			
	35		, ,			
	36 37	OUT name	A unique specifier for the actual (as opposed to vir- tual) node.			
	38	OUT resultlen	Length (in printable characters) of the result returned			
	39 40		in name			
	41					
ticket-248T.	42	int MP1_Get_processor_	name(char *name, int *resultlen)			
	43	-	(name, resultlen, ierror) BIND(C)			
	44		MAX_PROCESSOR_NAME), INTENT(OUT) :: name			
	45	INTEGER, INTENT(OU				
	46 47	INTEGER, OPTIONAL,				
	48	MPI_GET_PROCESSOR_NAME	(NAME, RESULTLEN, IERROR)			

CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR

{void MPI::Get_processor_name(char* name, int& resultlen)(binding deprecated, see Section 15.2 }

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger [then]than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger [then] than MPI_MAX_PROCESSOR_NAME.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI requires or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (End of rationale.)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (End of advice to users.)

The constant MPI_BSEND_OVERHEAD provides an upper bound on the fixed overhead per message buffered by a call to MPI_BSEND (see Section 3.6.1).

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the MPI_WIN_LOCK and MPI_WIN_UNLOCK functions to windows allocated in such memory (see Section 11.5.3.)

MPI_ALLC	DC_MEM(size, info, baseptr)		42
IN	size	size of memory segment in bytes (non-negative inte-	43
		ger)	44
IN	info	info argument (handle)	45
		into argument (nandie)	46
OUT	baseptr	pointer to beginning of memory segment allocated	47
			48

MPL ALLOC MEM(size, info, baseptr)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

56

 $\overline{7}$

8

9

10

23

 24

25

26

28

29 30 31

32 33

34

35

36

37

38

39

40 41

11 1213 14 ticket207. 15 ticket 207. 16 ticket 207. ¹⁷ ticket207. 18 19 202122

²⁷ ticket229.2.

```
1
                 int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
ticket-248T.
                MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
            3
                     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
            4
                     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
           5
                     TYPE(MPI_Info), INTENT(IN) :: info
           6
                     TYPE(C_PTR), INTENT(OUT) :: baseptr
            7
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            8
           9
                MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
           10
                     INTEGER INFO, IERROR
           11
                     INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
ticket229.1. 12
                 {void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info)(binding
           13
                                deprecated, see Section 15.2 }
           14
ticket229.1.
           15
                     If the Fortran compiler provides TYPE(C_PTR), then the following interface must be
ticket229.1.
           16
                 provided in the mpi module and should be provided in mpif.h through overloading, i.e., with
           17
                 the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR,
           18
                 but with a different linker name:
           19
                 INTERFACE MPI_ALLOC_MEM
           20
                     SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
           21
                         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
           22
           23
                         INTEGER :: INFO, IERROR
           24
                         INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
           25
                         TYPE(C_PTR) :: BASEPTR
           26
                     END SUBROUTINE
           27
                 END INTERFACE
           28
                     The linker name base of this overloaded function is MPI_ALLOC_MEM_CPTR. The
           29
                 implied linker names are described in Section 16.2.5 on page 651.
           30
                     The info argument can be used to provide directives that control the desired location
           31
                 of the allocated memory. Such a directive does not affect the semantics of the call. Valid
           32
                 info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL
           33
                 is always valid.
           34
                     The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
           35
                 to indicate it failed because memory is exhausted.
           36
           37
           38
                MPI_FREE_MEM(base)
           39
                  IN
                            base
                                                       initial address of memory segment allocated by
           40
                                                       MPI_ALLOC_MEM (choice)
           41
           42
           43
                 int MPI_Free_mem(void *base)
ticket229.2. 44
ticket-248T. 45
                MPI_Free_mem(base, ierror) BIND(C)
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
           46
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           47
           48
                MPI_FREE_MEM(BASE, IERROR)
```

<type> BASE(*) INTEGER IERROR</type>		1 2
		3
<pre>{void MPI::Free_mem(void *base)(binding deprecate</pre>	ed, see Section 15.2) }	4
The function MPI_FREE_MEM may return an er	ror code of class MPI_ERR_BASE to	5
indicate an invalid base argument.		6
0		7
Rationale. The C and C++ bindings of MPI_A	ALLOC_MEM and MPI_FREE_MEM	8
are similar to the bindings for the malloc and f	ree C library calls: a call to	9
MPI_Alloc_mem(, &base) should be paired with	a call to $MPI_Free_mem(base)$ (one	10
less level of indirection). Both arguments are de	eclared to be of same type void* so	11
as to facilitate type casting. The Fortran binding		12
bindings: the Fortran MPI_ALLOC_MEM call r	-	¹³ ticket 245 -Q.
pointer or the (integer valued) address of the all		14
of MPI_FREE_MEM is a choice argument, which	passes (a reference to) the variable	15
stored at that location. (End of rationale.)		16
		17
	A allocates special memory, then a	18
design similar to the design of C malloc and fre	· · ·	19
to find out the size of a memory segment, when		20
memory is used, MPI_ALLOC_MEM simply invo	21	
invokes free.		22
A call to MPI_ALLOC_MEM can be used in share		23
ory in a shared memory segment. (End of advice	to implementors.)	24
		$^{25}_{26}$ ticket245-Q.
Example 8.1 Example of use of MPI_ALLOC_ME	M, in Fortran with	27
TYPE(C_PTR) pointers. We assume 4-byte REALs.		28
	stad with INCLUDE Amaid ())	29
	<pre>nteed with INCLUDE 'mpif.h')</pre>	30
USE, INTRINSIC :: ISO_C_BINDING TYPE(C_PTR) :: p		31
REAL, DIMENSION(:,:), POINTER :: a	I no moment is allocated	32
INTEGER, DIMENSION(2) :: shape	! no memory is allocated	33
INTEGER, DIMENSION(2) Shape INTEGER(KIND=MPI_ADDRESS_KIND) :: size		34
shape = $(/100, 100/)$		35
shape = $(7100,1007)$ size = 4 * shape(1) * shape(2)	! assuming 4 bytes per REAL	36
CALL MPI_Alloc_mem(size, MPI_INFO_NULL, p, ierr		37
CALL C_F_POINTER(p, a, shape) ! intrinsic	· · · · · · · · · · · · · · · · · · ·	38
! in ISO_C_BIN		39
a(3,5) = 2.71;	DING	40
		41
CALL MPI_Free_mem(a, ierr)	! memory is freed	42
······································		43
		44

Example 8.2 []Example of use of MPI_ALLOC_MEM, in Fortran with [pointer support]non-standard *Cray-pointer*. We assume 4-byte REALs, and assume that these pointers are address-sized.

```
1
                    REAL A
            2
                    POINTER (P, A(100,100))
                                                   ! no memory is allocated
            3
                    [ticket245-Q.] INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
            4
                    [ticket245-Q.]SIZE = 4*100*100
            5
                    CALL MPI_ALLOC_MEM([ticket245-Q.] [4*100*100] SIZE, MPI_INFO_NULL, P, IERR)
            6
                    ! memory is allocated
            7
                    . . .
            8
                    A(3,5) = 2.71;
            9
                    . . .
            10
                    CALL MPI_FREE_MEM(A, IERR) ! memory is freed
ticket245-Q. 11
                      [Since standard Fortran does not support (C-like) pointers, this] This code is not Fortran
            12
ticket245-Q. 13
                  77 or Fortran 90 code. Some compilers (in particular, at the time of writing, g77 and Fortran
                  compilers for Intel) do may not support this code or need a special option, e.g., the GNU
ticket245-Q. 14
                  gFortran compiler needs -fcray-pointer.
ticket245-Q. 15
ticket229.2. <sub>17</sub>
                       Advice to implementors. Some compilers map Cray-pointers to address-sized integers,
                       some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's
            18
                       viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly
            19
ticket229.2.
                       with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to
            20
                       implementors.)
            21
            22
            23
                  Example 8.3 Same example, in C
            24
                    float (* f)[100][100];
            25
                    /* no memory is allocated */
            26
                    MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
            27
                    /* memory allocated */
            28
                    . . .
            29
                    (*f)[5][3] = 2.71;
            30
                    . . .
            31
                    MPI_Free_mem(f);
            32
            33
            34
                  8.3
                         Error Handling
            35
            36
                  An MPI implementation cannot or may choose not to handle some errors that occur during
            37
                  MPI calls. These can include errors that generate exceptions or traps, such as floating point
            38
                  errors or access violations. The set of errors that are handled by MPI is implementation-
            39
                  dependent. Each such error generates an MPI exception.
            40
                      The above text takes precedence over any text on error handling within this document.
            41
                  Specifically, text that states that errors will be handled should be read as may be handled.
            42
```

⁴² A user can associate error handlers to three types of objects: communicators, windows, ⁴³ and files. The specified error handling routine will be used for any MPI exception that occurs ⁴⁴ during a call to MPI for the respective object. MPI calls that are not related to any objects ⁴⁵ are considered to be attached to the communicator MPI_COMM_WORLD. The attachment ⁴⁶ of error handlers to objects is purely local: different processes may attach different error ⁴⁷ handlers to corresponding objects.

⁴⁸ Several predefined error handlers are available in MPI:

- **MPI_ERRORS_ARE_FATAL** The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.
- **MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM-_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non trivial MPI error handler.

After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does not necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (End of advice to implementors.)

29An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are 30 provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C and C++ have distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

34 An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER(function, errhandler), where XXX is, respectively, COMM, WIN, or FILE. 35

An error handler is attached to a communicator, window, or file by a call to MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, with matching XXX. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files. In C++, the predefined error handler MPI::ERRORS_THROW_EXCEPTIONS can also be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

47MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object 48 is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE

Unofficial Draft for Comment Only

1

2

3

4

5

6 $\overline{7}$

8

9

10

11

12

13

14

15

16

17

18

19

20

21

2223

 24

25

26

27

28

31

32

33

36

37

38 39

40 41

42

4344

	1 2 3	MPI_{COMM,WIN,FILE}_C	error handler returned from MPI_ERRHANDLER_GET or GET_ERRHANDLER to mark the error handler for deallocation. ilar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.
	4 5 6 7 8 9	attached to an object To do so, it is necess	<i>brs.</i> High-quality implementation should raise an error when was created by a call to MPI_XXX_CREATE_ERRHANDLER is of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. sary to maintain, with each error handler, information on the ted user function. (<i>End of advice to implementors.</i>)
	10 11	The syntax for these ca	alls is given below.
	12 13 14	8.3.1 Error Handlers for (Communicators
ticket252-W.	15 16		RHANDLER([function]comm_errhandler_fn, errhandler)
	17 18	IN [ticket252-W.][f	unction]comm_errhandler_fn user defined error handling procedure (function)
	19 20	OUT errhandler	MPI error handler (handle)
ticket252-W.	21 22 23		handler(MPI_Comm_errhandler_function h]comm_errhandler_fn, MPI_Errhandler *errhandler)
ticket-248T.	24 25 26	PROCEDURE(MPI_Comm_ TYPE(MPI_Errhandler	<pre>ller(comm_errhandler_fn, errhandler, ierror) BIND(C) errhandler_function) :: comm_errhandler_fn), INTENT(OUT) :: errhandler</pre>
ticket252-W.	27 28		<pre>INTENT(OUT) :: ierror DLER([FUNCTION]COMM_ERRHANDLER_FN, ERRHANDLER,</pre>
ticket252-W.	30 31	IERROR) EXTERNAL [FUNCTION] INTEGER ERRHANDLER,	
	32 33	{static MPI::Errhandler	
	34	C C	::Create_errhandler(MPI::Comm::Errhandler_function*
ticket252-W.		[function]	comm_errhandler_fn) (binding deprecated, see Section 15.2) }
	36 37	Creates an error hand	ler that can be attached to communicators. This function is
	38		DLER_CREATE, whose use is deprecated.
	39		l be, in C, a function of type $MPI_Comm_errhandler_function$, which
	40	is defined as	and an ill an fam at i an (MDT Gamma to int at a)
	41	typedei void MPI_Comm_e	errhandler_function(MPI_Comm *, int *,);
	42	The first argument is	the communicator in use. The second is the error code to be
	43	-	ne that raised the error. If the routine would have returned
	44		he error code returned in the status for the request that caused
	45 46		ked. The remaining arguments are "stdargs" arguments whose
	46 47		blementation-dependent. An implementation should clearly doc-
	48	0	ldresses are used so that the handler may be written in Fortran. Handler_function, whose use is deprecated.

With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be of the form:								
					ABSTRACT INTERFACE SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)			
TYPE(MPI_C	omm) :: comm		5					
INTEGER ::	error_code		7					
			8					
[In Fortran]With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN should be of the form:								
					SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)			
INTEGER COMM, ERROR_CODE			12					
			13					
In C++, the user routine should be of the form:								
<pre>{typedef void MPI::Comm::Errhandler_function(MPI::Comm &, int *,);</pre>								
					(00	taing acprecated, see		17
			18					
<i>Rationale.</i> The variable argument list is provided because it provides an ISO- standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (<i>End of rationale.</i>)								
					Advice to use	ore A nowly creat	ed communicator inherits the error handler that	22
							ommunicator. In particular, the user can specify	23
	-		24 25					
a "global" error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (<i>End of advice to users.</i>)								
)			27 28
			29					
			30					
MPI_COMM_SET_	ERRHANDLER(comm	, errhandler)	31					
INOUT comm		communicator (handle)	32					
IN errhand	ler	new error handler for communicator (handle)	33					
			34					
int MPT Comm set	errhandler(MPT Cc	mm comm, MPI_Errhandler errhandler)	35					
			$_{36}$ ticket-248T.					
MPI_Comm_set_err	37							
	m), INTENT(IN) ::		38					
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler								
INTEGER, OPT	40							
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR								
					<pre>{void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler)(binding</pre>			
dep	recureu, see Section 1	J.4] j	45					
Attaches a new error handler to a communicator. The error handler must be either								
a predefined error handler, or an error handler created by a call to								
			48					

```
360
                                              CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT
            1
                 MPI_COMM_CREATE_ERRHANDLER. This call is identical to MPI_ERRHANDLER_SET,
            \mathbf{2}
                 whose use is deprecated.
            3
            4
                 MPI_COMM_GET_ERRHANDLER(comm, errhandler)
            5
            6
                   IN
                            comm
                                                        communicator (handle)
            7
                   OUT
                            errhandler
                                                        error handler currently associated with communicator
            8
                                                        (handle)
            9
            10
                 int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
           11
ticket-248T.
            12
                 MPI_Comm_get_errhandler(comm, errhandler, ierror) BIND(C)
            13
                     TYPE(MPI_Comm), INTENT(IN) :: comm
            14
                     TYPE(MPI_Errhandler), INTENT(OUT) ::
                                                                errhandler
            15
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            16
                 MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
            17
                      INTEGER COMM, ERRHANDLER, IERROR
            18
            19
                 {MPI::Errhandler MPI::Comm::Get_errhandler() const(binding deprecated, see
           20
                                Section 15.2 }
           21
                     Retrieves the error handler currently associated with a communicator. This call is
           22
                 identical to MPI_ERRHANDLER_GET, whose use is deprecated.
           23
                     Example: A library function may register at its entry point the current error handler
           24
                 for a communicator, set its own private error handler for this communicator, and restore
           25
                 before exiting the previous error handler.
            26
           27
                 8.3.2 Error Handlers for Windows
            28
            29
            30
ticket252-W. ^{31}
                 MPI_WIN_CREATE_ERRHANDLER([function]win_errhandler_fn, errhandler)
            32
                             [ticket252-W.][function]win_errhandler_fn user defined error handling procedure (func-
                   IN
           33
                                                        tion)
           34
                   OUT
                            errhandler
                                                        MPI error handler (handle)
           35
           36
            37
                 int MPI_Win_create_errhandler(MPI_Win_errhandler_function
                                *[function]win_errhandler_fn, MPI_Errhandler *errhandler)
ticket252-W. <sup>38</sup>
ticket-248T. 39
                 MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror) BIND(C)
            40
                     PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
           41
                     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
           42
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           43
                 MPI_WIN_CREATE_ERRHANDLER([FUNCTION]WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
ticket252-W. 44
ticket252-W. <sup>45</sup>
                     EXTERNAL [FUNCTION] WIN_ERRHANDLER_FN
            46
                     INTEGER ERRHANDLER, IERROR
           47
           48
```

{static MPI::Err MPI	¹ ² ³ ticket252-W.		
Creates an err should be, in C, a t typedef void MPI	4 5 6 7		
The first argu	8		
With the Fortran m	$^{9}_{10}$ ticket230-B.		
ABSTRACT INTERFA	$^{10}_{11}$ ticket-248T.		
	12		
<pre>SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C) TYPE(MPI_Win) :: win</pre>			13
INTEGER ::	INTEGER :: error_code		
			15
[In Fortran]With t WIN_ERRHANDLE	¹⁶ ticket230-B. ¹⁷ ticket230-B.		
SUBROUTINE WIN_E	18		
INTEGER WIN, ERROR_CODE			19
	20 21		
In C++, the user \mathbf{r}	22		
{typedef void MF	23		
(bi	24		
			25
			26
MPI_WIN_SET_ER	27		
INOUT win		window (handle)	28
IN errhand	dler	new error handler for window (handle)	29 30
			31
int MPI_Win_set_	errhandler(MPI_Wi	n win, MPI_Errhandler errhandler)	32
	$_{33}$ ticket-248T.		
MPI_Win_set_errh TYPE(MPI_Win	34		
TYPE(MPI_WII	35		
INTEGER, OPT	36		
	37		
MPI_WIN_SET_ERRH	38 39		
INTEGER WIN,	40		
{void MPI::Win::	41		
deprecated, see Section 15.2) }			42
Attaches a ne	43		
defined error hand	44		
MPI_WIN_CREATE	45		
			46 47
			48

1 MPI_WIN_GET_ERRHANDLER(win, errhandler) 2 IN win window (handle) 3 OUT errhandler error handler currently associated with window (han-4 dle) 56 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler) ticket-248T. MPI_Win_get_errhandler(win, errhandler, ierror) BIND(C) 9 TYPE(MPI_Win), INTENT(IN) :: win 10 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR) 14INTEGER WIN, ERRHANDLER, IERROR 15{MPI::Errhandler MPI::Win::Get_errhandler() const(binding deprecated, see 16Section 15.2 } 17 18 Retrieves the error handler currently associated with a window. 19 208.3.3 Error Handlers for Files 2122 23ticket252-W. MPI_FILE_CREATE_ERRHANDLER([function]file_errhandler_fn, errhandler) 24 IN [ticket252-W.][function]file_errhandler_fn user defined error handling procedure (func-2526tion) 27OUT errhandler MPI error handler (handle) 28 29 int MPI_File_create_errhandler(MPI_File_errhandler_function 30 ticket252-W. *[function]file_errhandler_fn, MPI_Errhandler *errhandler) 31ticket-248T. 32 MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C) PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn 33 34 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 ticket252-W. MPI_FILE_CREATE_ERRHANDLER([FUNCTION]FILE_ERRHANDLER_FN, ERRHANDLER, 37 IERROR) ticket252-W. EXTERNAL [FUNCTION] FILE_ERRHANDLER_FN 30 INTEGER ERRHANDLER, IERROR 40 41 {static MPI::Errhandler 42MPI:::File::Create_errhandler(MPI::File::Errhandler_function* ticket252-W. 43 [function]file_errhandler_fn) (binding deprecated, see Section 15.2) } 44 Creates an error handler that can be attached to a file object. The user routine should 45be, in C, a function of type MPI_File_errhandler_function, which is defined as 46 typedef void MPI_File_errhandler_function(MPI_File *, int *, ...); 4748 The first argument is the file in use, the second is the error code to be returned.

With the Fortran mpi_f08 module, the user routine file_errhandler_fn should be of the form:				
ABSTRACT INTERFACE				
SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)				
TYPE(MPI_File) :: file		5		
INTEGER :: error_code	6			
In Fortron With the Fortron mpi modu	le and muif h the user routine	⁷ ticket230-B.		
[In Fortran]With the Fortran mpi module and mpif.h, the user routine FILE_ERRHANDLER_FN should be of the form:				
SUBROUTINE FILE_ERRHANDLER_FUNCTION	10			
INTEGER FILE, ERROR_CODE	11 12			
In C++, the user routine should be of th	13			
	14			
{typedef void MPI::File::Errhandle: (binding deprecated, see S	15			
(binaing aeprecatea, see s	<i>Section</i> 15.2)	16		
		17		
MPI_FILE_SET_ERRHANDLER(file, errha	ndler)	18		
		19		
INOUT file	file (handle)	20		
IN errhandler	new error handler for file (handle)	21 22		
		23		
<pre>int MPI_File_set_errhandler(MPI_Fil</pre>	le file, MPI_Errhandler errhandler)	24 ticket-248T.		
MPI_File_set_errhandler(file, errh	andler, ierror) BIND(C)	25		
TYPE(MPI_File), INTENT(IN) ::	file	26		
TYPE(MPI_Errhandler), INTENT(I	N) :: errhandler	27		
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	28		
MPI_FILE_SET_ERRHANDLER(FILE, ERRH	ANDLER, IERROR)	29		
INTEGER FILE, ERRHANDLER, IERR		30 31		
<pre>{void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler)(binding</pre>				
deprecated, see Section 1	32 33			
L /		34		
	le. The error handler must be either a predefined	35		
error handler, or an error handler created	by a call to MPI_FILE_CREATE_ERRHANDLER.	36		
		37		
MPI_FILE_GET_ERRHANDLER(file, errhandler)				
IN file	file (handle)	39		
OUT errhandler	error handler currently associated with file (handle)	40 41		
oor ernandier	error handler currently associated with me (handle)	42		
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)				
MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)				
TYPE(MPI_File), INTENT(IN) :: file				
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
INTEGER, OFFICIAL, INTENT(U01)	101101	48		

```
1
                 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
            \mathbf{2}
                     INTEGER FILE, ERRHANDLER, IERROR
            3
                 {MPI::Errhandler MPI::File::Get_errhandler() const(binding deprecated, see
            4
                                Section 15.2 }
            5
            6
                     Retrieves the error handler currently associated with a file.
            7
            8
                 8.3.4 Freeing Errorhandlers and Retrieving Error Strings
            9
            10
           11
                 MPI_ERRHANDLER_FREE( errhandler )
           12
                   INOUT
                            errhandler
                                                        MPI error handler (handle)
           13
           14
            15
                 int MPI_Errhandler_free(MPI_Errhandler *errhandler)
ticket-248T. 16
                 MPI_Errhandler_free(errhandler, ierror) BIND(C)
            17
                     TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
            18
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            19
           20
                 MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
           21
                      INTEGER ERRHANDLER, IERROR
           22
                 {void MPI::Errhandler::Free()(binding deprecated, see Section 15.2)}
           23
           ^{24}
                     Marks the error handler associated with errhandler for deallocation and sets errhandler
           25
                 to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects
           26
                 associated with it (communicator, window, or file) have been deallocated.
           27
           28
                 MPI_ERROR_STRING( errorcode, string, resultlen )
           29
           30
                   IN
                            errorcode
                                                        Error code returned by an MPI routine
           31
                   OUT
                            string
                                                        Text that corresponds to the errorcode
           32
                   OUT
                            resultlen
                                                        Length (in printable characters) of the result returned
           33
           34
                                                        in string
           35
           36
                 int MPI_Error_string(int errorcode, char *string, int *resultlen)
ticket-248T. 37
                 MPI_Error_string(errorcode, string, resultlen, ierror) BIND(C)
           38
                     INTEGER, INTENT(IN) :: errorcode
           39
                     CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
            40
                     INTEGER, INTENT(OUT) :: resultlen
           41
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           42
           43
                 MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
           44
                     INTEGER ERRORCODE, RESULTLEN, IERROR
           45
                     CHARACTER*(*) STRING
           46
                 {void MPI::Get_error_string(int errorcode, char* name,
           47
                                int& resultlen) (binding deprecated, see Section 15.2) }
           48
```

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

8.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes are shown in Table 8.1 and Table 8.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error string associated with an error class. An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

$$0 = MPI_SUCCESS < MPI_ERR_... \le MPI_ERR_LASTCODE.$$

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

MPI_ERROR_CLASS(errorcode, errorclass)

IN	errorcode	Error code returned by an MPI routine	37
		·	38
OUT	errorclass	Error class associated with errorcode	39
			40
int MPI_	Error_class(int	errorcode, int *errorclass)	⁴¹ ticket-248T.
MPI Erro	or class(errorcod	le, errorclass, ierror) BIND(C)	42
	GER, INTENT(IN)		43
	GER, INTENT(OUT)		44
		INTENT(OUT) :: ierror	45
	GLIG, OF ITOWAL, I		46
MPI_ERRC	R_CLASS (ERRORCOD	DE, ERRORCLASS, IERROR)	47
INTE	GER ERRORCODE, E	RRORCLASS, IERROR	48

Unofficial Draft for Comment Only

 $\overline{7}$

 24

1		
2	MPI_SUCCESS	No error
3	MPI_ERR_BUFFER	Invalid buffer pointer
4	MPI_ERR_COUNT	Invalid count argument
5	MPI_ERR_TYPE	Invalid datatype argument
6	MPI_ERR_TAG	Invalid tag argument
7	MPI_ERR_COMM	Invalid communicator
8	MPI_ERR_RANK	Invalid rank
9		
10	MPI_ERR_REQUEST	Invalid request (handle) Invalid root
11	MPI_ERR_ROOT	Invalid group
12	MPI_ERR_GROUP	
13	MPI_ERR_OP	Invalid operation
14	MPI_ERR_TOPOLOGY	Invalid topology
15	MPI_ERR_DIMS	Invalid dimension argument
16	MPI_ERR_ARG	Invalid argument of some other kind
17	MPI_ERR_UNKNOWN	Unknown error
18	MPI_ERR_TRUNCATE	Message truncated on receive
19	MPI_ERR_OTHER	Known error not in this list
20	MPI_ERR_INTERN	Internal MPI (implementation) error
21	MPI_ERR_IN_STATUS	Error code is in status
22	MPI_ERR_PENDING	Pending request
23	MPI_ERR_KEYVAL	Invalid keyval has been passed
24	MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory
25		is exhausted
26	MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
27	MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
28	MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
29	MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
30	MPI_ERR_SPAWN	Error in spawning processes
31	MPI_ERR_PORT	Invalid port name passed to
32		MPI_COMM_CONNECT
33	MPI_ERR_SERVICE	Invalid service name passed to
34		MPI_UNPUBLISH_NAME
35	MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
36		
37	MPI_ERR_WIN	Invalid win argument Invalid size argument
38	MPI_ERR_SIZE	Invalid disp argument
39	MPI_ERR_DISP MPI_ERR_INFO	Invalid info argument
40	MPI_ERR_LOCKTYPE	Invalid locktype argument
41	MPI_ERR_ASSERT	Invalid assert argument
42	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
43		0
44	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
45		
46	Table 8.1	1: Error classes (Part 1)
47		. ,
48		

MPI_ERR_FILE	Invalid file handle	1
MPI_ERR_NOT_SAME	Collective argument not identical on all	2
	processes, or collective routines called in	3
	a different order by different processes	4
MPI_ERR_AMODE	Error related to the amode passed to	5
	MPI_FILE_OPEN	6
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	7
	MPI_FILE_SET_VIEW	8
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	9
	a file which supports sequential access only	10
MPI_ERR_NO_SUCH_FILE	File does not exist	11
MPI_ERR_FILE_EXISTS	File exists	12
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	13
MPI_ERR_ACCESS	Permission denied	14
MPI_ERR_NO_SPACE	Not enough space	15
MPI_ERR_QUOTA	Quota exceeded	16
MPI_ERR_READ_ONLY	Read-only file or file system	17
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	18
	the file is currently open by some process	19
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	20
	tered because a data representation identi-	21
	fier that was already defined was passed to	22
	MPI_REGISTER_DATAREP	22
		23 24
MPI_ERR_CONVERSION	An error occurred in a user supplied data	
	conversion function.	25
MPI_ERR_IO	Other I/O error	26
MPI_ERR_LASTCODE	Last error code	27
		28
Table 8.2: Erre	or classes (Part 2)	29
		30
(int MDT, det annual lass (int annual)	(1, 1) (bin dimension dominants down $(2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$	31
{int MP1::Get_error_class(int errorc	ode) (binding deprecated, see Section 15.2) }	32
The function MPI_ERROR_CLASS ma	ps each standard error code (error class) onto	33
itself.		34
		35
		36
8.5 Error Classes, Error Codes, an	id Error Handlers	37
TT		38
· · · · ·	on top of an existing MPI implementation, and	39
	odes and classes. An example of such a library	40
	er 13 on page 511. For this purpose, functions	41
are needed to:		42
1. add a new error class to the ones an I	MPI implementation already knows.	43 44
2. associate error codes with this error of	class, so that MPI_ERROR_CLASS works.	44 45
3. associate strings with these error code	es, so that MPI_ERROR_STRING works.	$46 \\ 47$
4. invoke the error handler associated w	ith a communicator, window, or object.	47 48

48

1 Several functions are provided to do this. They are all local. No functions are provided $\mathbf{2}$ to free error classes or codes: it is not expected that an application will generate them in 3 significant numbers. 4 5MPI_ADD_ERROR_CLASS(errorclass) 6 $\overline{7}$ OUT errorclass value for the new error class (integer) 8 9 int MPI_Add_error_class(int *errorclass) ticket-248T. 10 MPI_Add_error_class(errorclass, ierror) BIND(C) 11 INTEGER, INTENT(OUT) :: errorclass 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) 15INTEGER ERRORCLASS, IERROR 1617{int MPI::Add_error_class() (binding deprecated, see Section 15.2) } 18 Creates a new error class and returns the value for it. 1920To avoid conflicts with existing error codes and classes, the value is set Rationale. 21by the implementation and not by the user. (End of rationale.) 22 23A high-quality implementation will return the value for Advice to implementors. 24a new errorclass in the same deterministic way on all processes. (End of advice to 25*implementors.*) 2627Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass 28may not be returned on all processes that make this call. Thus, it is not safe to assume 29that registering a new error on a set of processes at the same time will yield the same 30 errorclass on all of the processes. However, if an implementation returns the new 31errorclass in a deterministic way, and they are always generated in the same order on 32 the same set of processes (for example, all processes), then the value will be the same. 33 However, even if a deterministic algorithm is used, the value can vary across processes. 34 This can happen, for example, if different but overlapping groups of processes make 35a series of calls. As a result of these issues, getting the "same" error on multiple 36 processes may not cause the same value of error code to be generated. (End of advice 37 to users.) 38 The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-39 defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is 4041 associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the 42current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or 43 equal to MPI_ERR_LASTCODE. 4445Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change 46unless the user calls a function to explicitly add an error class/code. In a multi-47

Unofficial Draft for Comment Only

threaded environment, the user must take extra care in assuming this value has not

1 changed. Note that error codes and error classes are not necessarily dense. A user $\mathbf{2}$ may not assume that each error class below MPI_LASTUSEDCODE is valid. (End of 3 advice to users.) 4 56 MPI_ADD_ERROR_CODE(errorclass, errorcode) 7 IN errorclass error class (integer) 8 9 OUT errorcode new error code to associated with errorclass (integer) 10 11 int MPI_Add_error_code(int errorclass, int *errorcode) 12 ticket-248T. MPI_Add_error_code(errorclass, errorcode, ierror) BIND(C) 13 14INTEGER, INTENT(IN) :: errorclass 15INTEGER, INTENT(OUT) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) 18 INTEGER ERRORCLASS, ERRORCODE, IERROR 19 {int MPI::Add_error_code(int errorclass) (binding deprecated, see Section 15.2) } 2021Creates new error code associated with errorclass and returns its value in errorcode. 22 23*Rationale.* To avoid conflicts with existing error codes and classes, the value of the 24 new error code is set by the implementation and not by the user. (*End of rationale.*) 2526Advice to implementors. A high-quality implementation will return the value for 27a new errorcode in the same deterministic way on all processes. (End of advice to 28*implementors.*) 2930 31MPI_ADD_ERROR_STRING(errorcode, string) 32 33 IN errorcode error code or class (integer) 34 IN text corresponding to errorcode (string) string 35 36 int MPI_Add_error_string(int errorcode, const char *string) 37 ticket 140. 38 ticket-248T. MPI_Add_error_string(errorcode, string, ierror) BIND(C) 39 INTEGER, INTENT(IN) :: errorcode 40 CHARACTER(LEN=*), INTENT(IN) :: string 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) 43 INTEGER ERRORCODE, IERROR 44 CHARACTER*(*) STRING 4546{void MPI::Add_error_string(int errorcode, const char* string)(binding 47deprecated, see Section 15.2 } 48

	3 4 5 6 7 8 9	than MPL the calling or C++. for an error It is error ≤ MPL_ER If MF	MAX_ERROR_S g language. T Trailing blanks orcode that alr neous to call M R_LASTCODE. PI_ERROR_STE	STRING character he length of the s will be stripped eady has a strin IPI_ADD_ERROF	error code or class. The string must be no more rs long. The length of the string is as defined in string does not include the null terminator in C d in Fortran. Calling MPI_ADD_ERROR_STRING g will replace the old string with the new string. R_STRING for an error code or class with a value hen no string has been set, it will return a empty 1 C++).
	10 11 12 13		* 0	356 describes th rs, files, and wind	e methods for creating and associating error han- dows.
	14	MPI_COM	1M_CALL_ERR	HANDLER (com	m, errorcode)
	15 16	IN	comm	× ×	communicator with error handler (handle)
	17	IN	errorcode		error code (integer)
	18				
ticket-248T.	19	int MPI_	Comm_call_er:	rhandler(MPI_C	comm comm, int errorcode)
010100 2401.		MPI_Comm	_call_errhan	dler(comm, err	orcode, ierror) BIND(C)
	22			INTENT(IN) ::	
	23			IN) :: errorc	
	24			L, INTENT(OUT)	
	26			DLER(COMM, ERR RORCODE, IERRO	ORCODE, IERROR) R
	27 28 29	$\{void MP$	I::Comm::Call Section		nt errorcode) const(binding deprecated, see
	30	This	function invok	es the error hand	dler assigned to the communicator with the error
					2 SUCCESS in C and C++ and the same value in
	00				sfully called (assuming the process is not aborted
	34	and the en	rror handler re	turns).	
	35	Adv	ice to users.	Users should n	ote that the default error handler is
	36			,	lling MPI_COMM_CALL_ERRHANDLER will abort
	37 38		-		error handler has not been changed for this com-
	39	mun user		le parent before	the communicator was created. (End of advice to
	40	user			
	41				
	42 43	MPI_WIN	_CALL_ERRHA	ANDLER (win, er	rorcode)
	43 44	IN	win	,	window with error handler (handle)
	45	IN	errorcode		error code (integer)
	46				and some (moder)
ticket-248T.	47 48	int MPI_	Win_call_err	handler(MPI_Wi	n win, int errorcode)

		in, errorcode, ierror) BIND(C)	1
	TYPE(MPI_Win), INTENT		2
	<pre>INTEGER, INTENT(IN) :</pre>		3
	INTEGER, OPTIONAL, INT	TENT(OUT) :: ierror	4
MPI	WIN_CALL_ERRHANDLER(WI	IN. ERRORCODE. IERROR)	5
	INTEGER WIN, ERRORCODI		6
			7
{voi		andler(int errorcode) const(binding deprecated, see	8
	Section 15.2)	}	9 10
,	This function invokes the	error handler assigned to the window with the error code	10
		S MPI_SUCCESS in C and $C++$ and the same value in IERROR	12
		ssfully called (assuming the process is not aborted and the	13
	handler returns).		14
	,		15
		th communicators, the default error handler for windows is	16
	MPI_ERRORS_ARE_FATAL	(End of advice to users.)	17
			18
			19
MPI_	_FILE_CALL_ERRHANDLE	ER (fh, errorcode)	20
IN	fh	file with error handler (handle)	21
			22
IN	errorcode	error code (integer)	23
			24
			24
int	MPI_File_call_errhand	ler(MPI_File fh, int errorcode)	05
			$^{24}_{25}$ ticket-248T.
MPI_	File_call_errhandler(fh, errorcode, ierror) BIND(C)	25 ticket-248T.
MPI_		fh, errorcode, ierror) BIND(C) T(IN) :: fh	$^{25}_{26}$ ticket-248T.
MPI_	File_call_errhandler(: TYPE(MPI_File), INTEN	fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode	²⁵ ticket-248T. ²⁶
MPI_	File_call_errhandler(TYPE(MPI_File), INTEN INTEGER, INTENT(IN) : INTEGER, OPTIONAL, IN	fh, errorcode, ierror) BIND(C) F(IN) :: fh : errorcode FENT(OUT) :: ierror	²⁵ ticket-248T. ²⁶ ²⁷ ²⁸
MPI_	File_call_errhandler(TYPE(MPI_File), INTEN INTEGER, INTENT(IN) : INTEGER, OPTIONAL, IN FILE_CALL_ERRHANDLER(I	fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR)	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹
MPI_	File_call_errhandler(TYPE(MPI_File), INTEN INTEGER, INTENT(IN) : INTEGER, OPTIONAL, IN	fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR)	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰
MPI_ MPI_	File_call_errhandler(TYPE(MPI_File), INTEN INTEGER, INTENT(IN) : INTEGER, OPTIONAL, IN FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE	fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR)	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹
MPI_ MPI_	File_call_errhandler(TYPE(MPI_File), INTEN INTEGER, INTENT(IN) : INTEGER, OPTIONAL, IN FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³²
MPI_ MPI_ {voi	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see }</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³
MPI_ MPI_ {voi	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied.</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴
<pre>MPI_ MPI_ {voi This</pre>	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied.</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errh Section 15.2)] This function invokes the e function returns MPI_SUC handler was successfully ller returns).</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error errors on communicators and windows, the default behavior RRORS_RETURN. (End of advice to users.)</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴²
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error errors on communicators and windows, the default behavior RRORS_RETURN. (End of advice to users.) are warned that handlers should not be called recursively</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴² ⁴³
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error errors on communicators and windows, the default behavior RRORS_RETURN. (End of advice to users.) s are warned that handlers should not be called recursively _ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴² ⁴³ ⁴⁴
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error errors on communicators and windows, the default behavior RRORS_RETURN. (End of advice to users.) s are warned that handlers should not be called recursively _ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or NDLER. Doing this can create a situation where an infinite</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴² ⁴³ ⁴⁴ ⁴⁵
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error errors on communicators and windows, the default behavior RRORS_RETURN. (End of advice to users.) s are warned that handlers should not be called recursively _ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or NDLER. Doing this can create a situation where an infinite is can occur if MPI_COMM_CALL_ERRHANDLER,</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴² ⁴³ ⁴⁴ ⁴⁵ ⁴⁶
MPI_ MPI_ {voi This error	<pre>File_call_errhandler(: TYPE(MPI_File), INTENT INTEGER, INTENT(IN) : INTEGER, OPTIONAL, INT FILE_CALL_ERRHANDLER(I INTEGER FH, ERRORCODE d MPI::File::Call_errH</pre>	<pre>fh, errorcode, ierror) BIND(C) T(IN) :: fh : errorcode TENT(OUT) :: ierror FH, ERRORCODE, IERROR) , IERROR handler(int errorcode) const(binding deprecated, see } error handler assigned to the file with the error code supplied. CCESS in C and C++ and the same value in IERROR if the called (assuming the process is not aborted and the error errors on communicators and windows, the default behavior RRORS_RETURN. (End of advice to users.) s are warned that handlers should not be called recursively _ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or NDLER. Doing this can create a situation where an infinite</pre>	 ²⁵ ticket-248T. ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴² ⁴³ ⁴⁴ ⁴⁵

371

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (End of advice to users.)

Timers and Synchronization 8.6

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers. See also Section 2.6.5on page 23.

MPI_WTIME()

```
double MPI_Wtime(void)
18
```

```
DOUBLE PRECISION MPI_Wtime() BIND(C)
```

```
21
     DOUBLE PRECISION MPI_WTIME()
```

```
22
     {double MPI::Wtime()(binding deprecated, see Section 15.2) }
23
```

MPI_WTIME returns a floating-point number of seconds, representing elapsed wallclock time since some time in the past.

26The "time in the past" is guaranteed not to change during the life of the process. 27The user is responsible for converting large numbers of seconds to other units if they are preferred. 28

29This function is portable (it returns seconds, not "ticks"), it allows high-resolution, 30 and carries no unnecessary baggage. One would use it like this:

```
^{31}
                 {
           32
                    double starttime, endtime;
           33
                    starttime = MPI_Wtime();
           34
                      .... stuff to be timed
           35
                                                  . . .
                                = MPI_Wtime();
                    endtime
           36
                    printf("That took %f seconds\n",endtime-starttime);
           37
                 }
           38
           39
                     The times returned are local to the node that called them. There is no requirement
            40
                 that different nodes return "the same time." (But see also the discussion of
           41
                 MPI_WTIME_IS_GLOBAL).
           42
           43
           44
                 MPI_WTICK()
           45
            46
                 double MPI_Wtick(void)
ticket-248T. 47
                 DOUBLE PRECISION MPI_Wtick() BIND(C)
            48
```

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10

11

12

13

141516

17

19

20

 24

```
ticket-248T.
```

DOUBLE PRECISION MPI_WTICK()

{double MPI::Wtick() (binding deprecated, see Section 15.2) }

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3} .

8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

```
MPI_INIT()
```

```
int MPI_Init(int *argc, char ***argv)
MPI_Init(ierror) BIND(C)
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_INIT(IERROR)
```

```
INTEGER IERROR
```

```
{void MPI::Init(int& argc, char**& argv)(binding deprecated, see Section 15.2) }
```

{void MPI::Init()(binding deprecated, see Section 15.2) }

All MPI programs must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are MPI_GET_VERSION, []MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, [and] MPI_FINALIZED[], and any function with the prefix MPI_T_* (within the constraints for functions with this prefix listed in Section 14.3.4). The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    /* parse arguments */
    /* main program    */
    MPI_Finalize();    /* see below */
}
```

1

 $\mathbf{2}$

3

4 5

6

7

8 9

10 11

12

13

14

15

16

17

18

19 20 21

22

25

26

27

28 29

30 31

32

33

34

35

39 40

41

42

43 44

45

46 47

```
^{23}_{24} ticket-248T.
```

```
<sub>36</sub> ticket204.
<sub>37</sub> ticket266.
```

```
<sup>37</sup> ticket266
```

1 The Fortran version takes only IERROR. $\mathbf{2}$ Conforming implementations of MPI are required to allow applications to pass NULL 3 for both the argc and argv arguments of main in C and C++. In C++, there is an alternative 4 binding for MPI::Init that does not have these arguments at all. 56 Rationale. In some applications, libraries may be making the call to MPI_Init, and may not have access to argc and argv from main. It is anticipated 7 that applications requiring special information about the environment or information 8 supplied by mpiexec can get that information from environment variables. (End of 9 rationale.) 10 11 1213MPI_FINALIZE() 1415int MPI_Finalize(void) ticket-248T. 16 MPI_Finalize(ierror) BIND(C) 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19MPI_FINALIZE(IERROR) 20INTEGER IERROR 21{void MPI::Finalize()(binding deprecated, see Section 15.2) } 2223This routine cleans up all MPI state. Each process must call MPI_FINALIZE before 24 it exits. Unless there has been a call to MPI_ABORT, each process must ensure that all 25pending nonblocking communications are (locally) complete before calling MPI_FINALIZE. 26Further, at the instant at which the last process calls MPI_FINALIZE, all pending sends 27must be matched by a receive, and all pending receives must be matched by a send. 28For example, the following program is correct: 29 30 Process 0 Process 1 31_____ _____ 32 MPI_Init(); MPI_Init(); 33 MPI_Send(dest=1); MPI_Recv(src=0); 34 MPI_Finalize(); MPI_Finalize(); 35 Without the matching receive, the program is erroneous: 36 37 Process 0 Process 1 38 _____ _____ 39 MPI_Init(); MPI_Init(); 40 MPI_Send (dest=1); 41 MPI_Finalize(); MPI_Finalize(); 4243 A successful return from a blocking communication operation or from MPI_WAIT or 44MPI_TEST tells the user that the buffer can be reused and means that the communication 45is completed by the user, but does not guarantee that the local process has no more work 46to do. A successful return from MPI_REQUEST_FREE with a request handle generated by 47an MPI_ISEND nullifies the handle but provides no assurance of operation completion. The

⁴⁸ MPI_ISEND is complete only when it is known by some means that a matching receive has

it.

completed. MPI_FINALIZE guarantees that all local actions required by communications 1 2 the user has completed will, in fact, occur before it returns. 3 MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE 4 combined with some other verification of completion). 56 **Example 8.4** This program is correct: 7 8 rank 0 rank 1 9 _____ 10 11 MPI_Isend(); MPI_Recv(); 12MPI_Request_free(); MPI_Barrier(); 13 MPI_Barrier(); MPI_Finalize(); 14MPI_Finalize(); exit(); 15exit(); 1617 **Example 8.5** This program is erroneous and its behavior is undefined: 18 19 rank 0 rank 1 20_____ 21. 22 MPI_Isend(); MPI_Recv(); 23 MPI_Request_free(); MPI_Finalize(); 24 MPI_Finalize(); exit(); 25exit(); 2627If no MPI_BUFFER_DETACH occurs between an MPI_BSEND (or other buffered send) 28and MPI_FINALIZE, the MPI_FINALIZE implicitly supplies the MPI_BUFFER_DETACH. 29**Example 8.6** This program is correct, and after the MPI_Finalize, it is as if the buffer 30 had been detached. 3132 rank 0 rank 1 33 34 35buffer = malloc(1000000); MPI_Recv(); 36 MPI_Buffer_attach(); MPI_Finalize(); 37 MPI_Bsend(); exit(); 38 MPI_Finalize(); 39 free(buffer); 40 exit(); 41 42Example 8.7 In this example, MPI_lprobe() must return a FALSE flag. 43 44MPI_Test_cancelled() must return a TRUE flag, independent of the relative order of execution of MPI_Cancel() in process 0 and MPI_Finalize() in process 1. 4546The MPI_lprobe() call is there to make sure the implementation knows that the "tag1" 47message exists at the destination, without being able to claim that the user knows about

```
48
```

1 $\mathbf{2}$ rank 0 rank 1 3 _____ _____ 4MPI_Init(); MPI_Init(); 5MPI_Isend(tag1); 6 MPI_Barrier(); MPI_Barrier(); 7 MPI_Iprobe(tag2); 8 MPI_Barrier(); MPI_Barrier(); 9 MPI_Finalize(); 10 exit(); 11MPI_Cancel(); 12MPI_Wait(); 13 MPI_Test_cancelled(); 14MPI_Finalize(); 15exit(); 1617 Advice to implementors. An implementation may need to delay the return from 18 MPI_FINALIZE until all potential future message cancellations have been processed. 19 One possible solution is to place a barrier inside MPI_FINALIZE (End of advice to 20*implementors.*) 2122 Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, 23 ticket204. 24 except for MPI_GET_VERSION, [MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, [and] ticket266. 25 MPI_FINALIZED], and any function with the prefix MPI_T_* (within the constraints for ticket266. $_{26}$ functions with this prefix listed in Section 14.3.4). Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process 27may continue local computations, or exit, without participating in further MPI communi-28 cation with other processes. MPI_FINALIZE is collective over all connected processes. If no 29 processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; 30 otherwise it is collective over the union of all processes that have been and continue to be 31connected, as explained in Section 10.5.4 on page 416. 32 33 Advice to implementors. Even though a process has completed all the communication 34it initiated, such communication may not yet be completed from the viewpoint of the 35 underlying MPI system. E.g., a blocking send may have completed, even though the 36 data is still buffered at the sender. The MPI implementation must ensure that a 37 process has completed any involvement in MPI communication before MPI_FINALIZE 38 returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause 39 an ongoing communication to fail. (End of advice to implementors.) 40 41 Although it is not required that all processes return from MPI_FINALIZE, it is required 42that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI 43 portion of the computation is over. In addition, in a POSIX environment, they may desire 44to supply an exit code for each process that returns from MPI_FINALIZE. 45

Example 8.8 The following illustrates the use of requiring that at least one process return
 and that it be known that process 0 is one of the processes that return. One wants code
 like the following to work no matter how many processes return.

			1
	MPI_Comm_rank(MPI_COMM_WORLD,	&myrank);	2
			3
	<pre>MPI_Finalize(); if (manual = 0) {</pre>		4 5
	<pre>if (myrank == 0) { mogultfile = feren("eutfil </pre>		6
	<pre>resultfile = fopen("outfil dump_results(resultfile);</pre>	e, w);	7
	fclose(resultfile);		8
	}		9
	exit(0);		10
			11
			12
MDI	_INITIALIZED(flag)		13
	ζ Ξ γ		14
01	JT flag	Flag is true if MPI_INIT has been called and false	15
		otherwise.	16
			17 18
int	MPI_Initialized(int *flag)		¹⁹ ticket-248T.
MPI_	Initialized(flag, ierror) BIND	(C)	20
	LOGICAL, INTENT(OUT) :: flag		21
	INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	22
MPI_INITIALIZED(FLAG, IERROR)			23
···· ±-	LOGICAL FLAG		24
	INTEGER IERROR		25
(1			26
{bod	1 MPI::Is_initialized()(binding	deprecated, see Section 15.2) }	27
	This routine may be used to determ	nine whether MPI_INIT has been called.	28 29
MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one			
			33
MPI	_ABORT(comm, errorcode)		34
IN	comm	communicator of tasks to abort	35
			36
IN	errorcode	error code to return to invoking environment	37
			38
int	MPI_Abort(MPI_Comm comm, int e	rrorcode)	$^{39}_{40}$ ticket-248T.
MPI_	Abort(comm, errorcode, ierror)	BIND(C)	
	TYPE(MPI_Comm), INTENT(IN) ::	comm	41
	<pre>INTEGER, INTENT(IN) :: errorc</pre>	ode	42 43
	INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	43
MPT	ABORT (COMM, ERRORCODE, IERROR)		45
	INTEGER COMM, ERRORCODE, IERRO		46
ſ.			47
<pre>{void MPI::Comm::Abort(int errorcode)(binding deprecated, see Section 15.2) }</pre>			48

This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a return errorcode from the main program.

⁵ It may not be possible for an MPI implementation to abort only the processes repre-⁶ sented by **comm** if this is a subset of the processes. In this case, the MPI implementation ⁷ should attempt to abort all the connected processes but should not abort any unconnected ⁸ processes. If no processes were spawned, accepted or connected then this has the effect of ⁹ aborting all the processes associated with MPI_COMM_WORLD.

Rationale. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI_COMM_WORLD. (*End of rationale.*)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (End of advice to users.)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

23 24 25

47 48

10 11

12

13

14 15

16

17

18 19 20

21

22

8.7.1 Allowing User Functions at Process Termination

26There are times in which it would be convenient to have actions happen when an MPI process 27finishes. For example, a routine may do initializations that are useful until the MPI job (or 28that part of the job that being terminated in the case of dynamically created processes) is 29finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF 30 with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent 31 of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function 32 to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that 33 they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then 34no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts 35 of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any 36 of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the 37 actions taken by MPI_FINALIZE is not specified. 38

Advice to implementors. Since attributes can be added from any supported language,
 the MPI implementation needs to remember the creating language so the correct
 callback is made. Implementations that use the attribute delete callback on
 MPI_COMM_SELF internally should register their internal callbacks before returning
 from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have
 portions of the MPI implementation shut down before the application-level callbacks
 are made. (End of advice to implementors.)

8.7.2 Determining Whether MPI Has Finished	1
One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been	2 3 4
finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following	5 6 7
function is needed:	8
MPI_FINALIZED(flag)	10
OUTflagtrue if MPI was finalized (logical)	11 12
<pre>int MPI_Finalized(int *flag)</pre>	¹³ , ticket-248T.
<pre>MPI_Finalized(flag, ierror) BIND(C) LOGICAL, INTENT(OUT) :: flag</pre>	16 11CKet-2401.
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17 18
MPI_FINALIZED(FLAG, IERROR) LOGICAL FLAG	19 20
INTEGER IERROR	21
<pre>{bool MPI::Is_finalized()(binding deprecated, see Section 15.2) }</pre>	22 23
This routine returns true if MPI_FINALIZE has completed. It is [legal]valid to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.	²⁴ ticket182. ²⁵ ²⁶
Advice to users. MPI is "active" and it is thus safe to call MPI functions if MPI_INIT has completed and MPI_FINALIZE has not completed. If a library has no other	27 28
way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is "active" in callback functions that are invalid during MPI_FINALIZE (End of advice to users)	29 30
that are invoked during MPI_FINALIZE. (End of advice to users.)	31 32
8.8 Portable MPI Process Startup	33 34
A number of implementations of MPI provide a startup command for MPI programs that is of the form	35 36
mpirun <mpirun arguments=""> <program> <program arguments=""></program></program></mpirun>	37 38
Separating the command to start the program from the program itself provides flexibility,	39
particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.	40 41
Having a standard startup mechanism also extends the portability of MPI programs one	42 43
step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such	44 45
a standard starup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead	46
of mpirun MPI specifies mpiexec.	47 48

1 While a standardized startup mechanism improves the usability of MPI, the range of $\mathbf{2}$ environments is so diverse (e.g., there may not even be a command line interface) that MPI 3 cannot mandate such a mechanism. Instead, MPI specifies an mpiexec startup command 4 and recommends but does not require it, as advice to implementors. However, if an im-5plementation does provide a command called **mpiexec**, it must be of the form described 6 below. 7 It is suggested that 8 mpiexec -n <numprocs> <program> 9 10 be at least one way to start <program> with an initial MPI_COMM_WORLD whose group 11 contains <numprocs> processes. Other arguments to mpiexec may be implementation-12dependent. 13 14Advice to implementors. Implementors, if they do provide a special startup command 15for MPI programs, are advised to give it the following form. The syntax is chosen in 16order that mpiexec be able to be viewed as a command-line version of 17 MPI_COMM_SPAWN (See Section 10.3.4). 18 Analogous to MPI_COMM_SPAWN, we have 19 20mpiexec -n <maxprocs> 21-soft < > 22 -host < > 23 -arch < > 24-wdir < > 25-path < > 26< -file > 27. . . 28 <command line> 29 30 31for the case where a single command line for the application program and its arguments 32 will suffice. See Section 10.3.4 for the meanings of these arguments. For the case 33 corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats: 34 Form A: 3536 mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... } 37 38 As with MPI_COMM_SPAWN, all the arguments are optional. (Even the $-n \times argu-$ 39 ment is optional; the default is implementation dependent. It might be 1, it might be 40 taken from an environment variable, or it might be specified at compile time.) The 41 names and meanings of the arguments are taken from the keys in the info argument 42to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments 43 as well. 44 45 Note that Form A, though convenient to type, prevents colons from being program 46 arguments. Therefore an alternate, file-based form is allowed: 47 Form B:

	1
<pre>mpiexec -configfile <filename></filename></pre>	2
	3
where the lines of <i><</i> filename> are of the form separated by the colons in Form A.	4
Lines beginning with '#' are comments, and lines may be continued by terminating	5
the partial line with $\langle \cdot \rangle$.	6
	7
Example 8.9 Start 16 instances of myprog on the current or default machine:	8
	9
mpiexec -n 16 myprog	10
	10
	12
Example 8.10 Start 10 processes on the machine called ferrari:	
	13
mpiexec -n 10 -host ferrari myprog	14
	15
	16
Example 8.11 Start three copies of the same program with different command-line	17
arguments:	18
	19
<pre>mpiexec myprog infile1 : myprog infile2 : myprog infile3</pre>	20
	21
Europeanle 8 13. Start the second program on five Sung and the stress program on 10	22
Example 8.12 Start the ocean program on five Suns and the atmos program on 10	23
RS/6000's:	24
	25
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos	26
	27
It is assumed that the implementation in this case has a method for choosing hosts of	28
the appropriate type. Their ranks are in the order specified.	29
	30
Example 8.13 Start the ocean program on five Suns and the atmos program on 10	31
RS/6000's (Form B):	32
	33
mpiexec -configfile myfile	34
	35
where myfile contains	36
	37
-n 5 -arch sun ocean	38
-n 10 -arch rs6000 atmos	39
	40
(End of advice to implementors.)	41
	42
	43
	44
	45
	46
	47
	48

Chapter 9

The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI_Info in C and Fortran with the mpi_f08 module, MPI::Info in C++, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

Keys have an implementation-defined maximum length of MPI_MAX_INFO_KEY, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI_MAX_INFO_VAL. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size MPI_MAX_INFO_KEY. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. MPI_MAX_INFO_VAL might be very large, so it might not be wise to declare a string of that size. (End of advice to users.)

When it is an argument to a nonblocking routine, info is parsed before that routine returns, so that it may be modified or freed immediately after return.

When the descriptions refer to a key or value as being a boolean, an integer, or a list, they mean the string representation of these types. An implementation may define its own rules for how info value strings are converted to other types, but to ensure portability, every implementation must support the following representations. Legal values for a boolean must

¹⁵ ticket231-C. ¹⁶ ticket231-C.

 24

 31

1 include the strings "true" and "false" (all lowercase). For integers, legal values must include $\mathbf{2}$ string representations of decimal values of integers that are within the range of a standard 3 integer type in the program. (However it is possible that not every legal integer is a legal 4 value for a given key.) On positive numbers, + signs are optional. No space may appear $\mathbf{5}$ between a + or - sign and the leading digit of a number. For comma separated lists, the 6 string must contain legal elements separated by commas. Leading and trailing spaces are 7stripped automatically from the types of info values described above and for each element of 8 a comma separated list. These rules apply to all info values of these types. Implementations 9 are free to specify a different interpretation for values of other info keys. 10 11MPI_INFO_CREATE(info) 1213 OUT info info object created (handle) 1415int MPI_Info_create(MPI_Info *info) ticket-248T. 16 MPI_Info_create(info, ierror) BIND(C) 17TYPE(MPI_Info), INTENT(OUT) :: info 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1920MPI_INFO_CREATE(INFO, IERROR) 21INTEGER INFO, IERROR 22{static MPI::Info MPI::Info::Create()(binding deprecated, see Section 15.2)} 23 24 MPI_INFO_CREATE creates a new info object. The newly created object contains no 25key/value pairs. 262728MPI_INFO_SET(info, key, value) 29INOUT info info object (handle) 30 IN key (string) key 31 32 IN value value (string) 33 34 ticket140. int MPI_Info_set(MPI_Info info, const char *key, const char *value) 35 ticket140. MPI_Info_set(info, key, value, ierror) BIND(C) ticket-248T. 36 TYPE(MPI_Info), INTENT(IN) :: info 37 CHARACTER(LEN=*), INTENT(IN) :: key, value 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40MPI_INFO_SET(INFO, KEY, VALUE, IERROR) 41 INTEGER INFO, IERROR 42CHARACTER*(*) KEY, VALUE 43 {void MPI::Info::Set(const char* key, const char* value) (binding deprecated, 44see Section 15.2 } 4546 MPI_INFO_SET adds the (key, value) pair to info, and overrides the value if a value for 47 the same key was previously set. key and value are null-terminated strings in C. In Fortran, 48

0	01	n key and value are stripped. If either key or value are larger	1 2
	,	, the errors $MPI_ERR_INFO_KEY$ or $MPI_ERR_INFO_VALUE$ are	2 3
raiseu, 16	espectively.		4
			5
MPI_INF	O_DELETE(info, key	/)	6
INOUT	info	info object (handle)	7
IN	key	key (string)	8
IIN	ney	Key (Sumg)	9
int MPI_	_Info_delete(MPI_	Info info, <mark>const</mark> char *key)	$^{10}_{11}$ ticket 140.
MPT Info	o delete(info, ke	y, ierror) BIND(C)	$_{12}$ ticket-248'
	E(MPI_Info), INTEN	·	13
	RACTER(LEN=*), INT		14
		NTENT(OUT) :: ierror	15
	O_DELETE(INFO, KEY		16
	EGER INFO, IERROR		17 18
	RACTER*(*) KEY		18 19
		(1, 1) $(1, 1)$ $(1, 2)$ $(1, 2)$ $(1, 2)$	20
-		<pre>const char* key)(binding deprecated, see Section 15.2) }</pre>	21
		etes a (key,value) pair from info. If key is not defined in info,	22
the call r	aises an error of clas	ss MPI_ERR_INFO_NOKEY.	23
			24
MPI_INF	O_GET(info, key, valu	uelen. value. flag)	25
IN	info	info object (handle)	26 27
			27 28
IN	key	key (string)	28
IN	valuelen	length of value arg (integer)	30
OUT	value	value (string)	31
OUT	flag	true if key defined, false if not (boolean)	32 33
int MPI_	-	o info, const char *key, int valuelen, char *value,	$^{34}_{_{35}}$ ticket140.
	int *flag)		³⁶ ticket-248
MPI_Info	o_get(info, key, [,]	valuelen, value, flag, ierror) BIND(C)	37
TYPE	E(MPI_Info), INTEN	NT(IN) :: info	38
	RACTER(LEN=*), INT		39
	EGER, INTENT(IN) :		40
		en), INTENT(OUT) :: value	41
	ICAL, INTENT(OUT) EGER OPTIONAL IN	:: flag NTENT(OUT) :: ierror	42
			43 44
		VALUELEN, VALUE, FLAG, IERROR)	44
	EGER INFO, VALUELE	-	46
	RACTER*(*) KEY, VA	ALUE	47
LUGI	ICAL FLAG		48

```
386
                                                                        CHAPTER 9. THE INFO OBJECT
            1
                  {bool MPI::Info::Get(const char* key, int valuelen, char* value)
            \mathbf{2}
                                  const(binding deprecated, see Section 15.2)
             3
                       This function retrieves the value associated with key in a previous call to
             4
                  MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,
            5
                  otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters
             6
                  available in value. If it is less than the actual size of the value, the value is truncated. In
             7
                  C, valuelen should be one less than the amount of allocated space to allow for the null
             8
                  terminator.
            9
                      If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
            10
            11
            12
                  MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
            13
                    IN
                              info
                                                            info object (handle)
            14
                    IN
                                                            key (string)
            15
                              key
            16
                    OUT
                              valuelen
                                                           length of value arg (integer)
            17
                    OUT
                              flag
                                                            true if key defined, false if not (boolean)
            18
            19
  ticket140. 20
                  int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
                                  int *flag)
            21
ticket-248
T<br/>. _{\scriptscriptstyle 22}
                  MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) BIND(C)
            23
                       TYPE(MPI_Info), INTENT(IN) :: info
            ^{24}
                       CHARACTER(LEN=*), INTENT(IN) :: key
            25
                       INTEGER, INTENT(OUT) :: valuelen
            26
                       LOGICAL, INTENT(OUT) :: flag
            27
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            28
                  MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
            29
                       INTEGER INFO, VALUELEN, IERROR
            30
                       LOGICAL FLAG
            ^{31}
                       CHARACTER*(*) KEY
            32
            33
                  {bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const(binding
            34
                                  deprecated, see Section 15.2 }
            35
                       Retrieves the length of the value associated with key. If key is defined, valuelen is set
            36
            37
                  to the length of its associated value and flag is set to true. If key is not defined, valuelen is
                  not touched and flag is set to false. The length returned in C or C++ does not include the
            38
                  end-of-string character.
            39
                      If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
            40
            41
            42
                  MPI_INFO_GET_NKEYS(info, nkeys)
            43
                    IN
                              info
                                                            info object (handle)
            44
            45
                    OUT
                              nkeys
                                                            number of defined keys (integer)
            46
            47
                  int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
            48
```

```
1
ticket-248T.
                 MPI_Info_get_nkeys(info, nkeys, ierror) BIND(C)
                                                                                                          \mathbf{2}
                     TYPE(MPI_Info), INTENT(IN) :: info
                                                                                                          3
                     INTEGER, INTENT(OUT) :: nkeys
                                                                                                          4
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                                          5
                                                                                                          6
                 MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
                                                                                                          7
                     INTEGER INFO, NKEYS, IERROR
                 {int MPI:::Info::Get_nkeys() const(binding deprecated, see Section 15.2) }
                                                                                                          9
                                                                                                          10
                     MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
                                                                                                          11
                                                                                                          12
                                                                                                          13
                 MPI_INFO_GET_NTHKEY(info, n, key)
                                                                                                          14
                   IN
                            info
                                                        info object (handle)
                                                                                                          15
                   IN
                                                        key number (integer)
                            n
                                                                                                          16
                                                                                                          17
                   OUT
                                                        key (string)
                            key
                                                                                                          18
                                                                                                          19
                 int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
                                                                                                          ^{20} ticket-248T.
                 MPI_Info_get_nthkey(info, n, key, ierror) BIND(C)
                                                                                                          21
                     TYPE(MPI_Info), INTENT(IN) :: info
                                                                                                          22
                     INTEGER, INTENT(IN) :: n
                                                                                                          23
                     CHARACTER(LEN=*), INTENT(OUT) :: key
                                                                                                          ^{24}
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                                          25
                                                                                                          26
                 MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
                                                                                                          27
                     INTEGER INFO, N, IERROR
                                                                                                          28
                     CHARACTER*(*) KEY
                                                                                                          29
                 {void MPI::Info::Get_nthkey(int n, char* key) const(binding deprecated, see
                                                                                                          30
                                Section 15.2 }
                                                                                                          31
                                                                                                          32
                     This function returns the nth defined key in info. Keys are numbered 0 \dots N-1 where
                                                                                                          33
                 N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and N-1 are
                                                                                                          34
                 guaranteed to be defined. The number of a given key does not change as long as info is not
                                                                                                          35
                 modified with MPI_INFO_SET or MPI_INFO_DELETE.
                                                                                                          36
                                                                                                          37
                                                                                                          38
                 MPI_INFO_DUP(info, newinfo)
                                                                                                          39
                   IN
                            info
                                                        info object (handle)
                                                                                                          40
                   OUT
                            newinfo
                                                        info object (handle)
                                                                                                          41
                                                                                                          42
                                                                                                          43
                 int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
                                                                                                          44 ticket-248T.
                 MPI_Info_dup(info, newinfo, ierror) BIND(C)
                                                                                                          45
                     TYPE(MPI_Info), INTENT(IN) :: info
                                                                                                          46
                     TYPE(MPI_Info), INTENT(OUT) :: newinfo
                                                                                                          47
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                                          48
```

```
1
                  MPI_INFO_DUP(INFO, NEWINFO, IERROR)
            \mathbf{2}
                       INTEGER INFO, NEWINFO, IERROR
            3
                  {MPI:::Info MPI:::Info:::Dup() const(binding deprecated, see Section 15.2) }
            4
            5
                       MPI_INFO_DUP duplicates an existing info object, creating a new object, with the
            6
                  same (key,value) pairs and the same ordering of keys.
            7
            8
                  MPI_INFO_FREE(info)
            9
            10
                    INOUT
                              info
                                                            info object (handle)
            11
            12
                  int MPI_Info_free(MPI_Info *info)
ticket-248T. 13
                  MPI_Info_free(info, ierror) BIND(C)
            14
                       TYPE(MPI_Info), INTENT(INOUT) :: info
            15
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            16
            17
                  MPI_INFO_FREE(INFO, IERROR)
            18
                       INTEGER INFO, IERROR
            19
                  {void MPI::Info::Free() (binding deprecated, see Section 15.2) }
            20
            21
                       This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is
            22
                  interpreted each time the info is passed to a routine. Changes to an info after return from
            23
                  a routine do not affect that interpretation.
            24
            25
            26
            27
            28
            29
            30
            ^{31}
            32
            33
            34
            35
            36
            37
            38
            39
            40
            41
            42
            43
            44
            45
            46
            47
            48
```

Chapter 10

Process Creation and Management

10.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allow for a variety of approaches to process management while placing minimal restrictions on the execution environment.

The MPI model for process creation allows both the creation of an initial set of processes related by their membership in a common MPI_COMM_WORLD and the creation and management of processes after an MPI application has been started. A major impetus for the later form of process creation comes from the PVM [24] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. assumes that resource control is provided externally — probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for including process management in MPI are both technical and practical. Important classes of message-passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM's capabilities for process and resource management. The lack of these features would be a practical stumbling block to migration.

The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.
- MPI must not take over operating system responsibilities. It should instead provide a

 24

 31

clean interface between an application and system software.

- MPI must guarantee communication determinism in the presense of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.

The process management model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI maintains a consistent concept of a communicator, regardless of how its members came into existence. A communicator is never changed once created, and it is always created using deterministic collective operations.

10.2 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not "start" the other.

22 23 24

10.2.1 Starting Processes

 $^{25}_{26}$ MPI applications may start new processes through an interface to an external process manager.

²⁷ MPI_COMM_SPAWN starts MPI processes and establishes communication with them,
 ²⁸ returning an intercommunicator. MPI_COMM_SPAWN_MULTIPLE starts several different
 ³⁰ binaries (or the same binary with different arguments), placing them in the same
 ³¹ MPI_COMM_WORLD and returning an intercommunicator.

MPI uses the existing group abstraction to represent processes. A process is identified by a (group, rank) pair.

10.2.2 The Runtime Environment

The MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one. Examples of such environments are:

 $40 \\ 41$

42

43

44

45

46 47 48

32

33 34

35

• MPP managed by a batch queueing system. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.

8

9

10

11

12

13

14 15

16 17

18

19

20

21

- Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a "virtual machine" out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.
- Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.
- Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API. An example of such an API would be the PVM task and machine management routines — pvm_addhosts, pvm_config, pvm_tasks, etc., possibly modified to return an MPI (group,rank) when possible. A Condor or PBS API would be another possibility.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.
- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.

391

1 2

3

4

5

6

 $\overline{7}$

8

9

10 11

12

13

14

15

16 17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41 42

43

44

45

46

47 48

1 • An attribute MPI_UNIVERSE_SIZE on MPI_COMM_WORLD tells a program how "large" $\mathbf{2}$ the initial runtime environment is, namely how many processes can usefully be started 3 in all. One can subtract the size of MPI_COMM_WORLD from this value to find out 4 how many processes might usefully be started in addition to those already running. 56 **Process Manager Interface** 10.3 7 8 10.3.1 Processes in MPI 9 10 A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group, rank) pair, since a process 11 may belong to several groups. 1213 1410.3.2 Starting Processes and Establishing Communication 15The following routine starts a number of MPI processes and establishes communication with 16them, returning an intercommunicator. 1718 Advice to users. It is possible in MPI to start a static SPMD or MPMD appli-19 cation by starting first one process and having that process start its siblings with 20MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of perfor-21mance. If possible, it is preferable to start all processes at once, as a single MPI 22 application. (End of advice to users.) 23242526MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, 27array_of_errcodes) 28IN command name of program to be spawned (string, significant 29 only at root) 30 IN arguments to command (array of strings, significant argv 31 only at root) 32 33 IN maxprocs maximum number of processes to start (integer, sig-34 nificant only at root) 35 IN info a set of key-value pairs telling the runtime system 36 where and how to start the processes (handle, signifi-37 cant only at root) 38 IN rank of process in which previous arguments are exroot 39 amined (integer) 40 41 IN comm intracommunicator containing group of spawning pro-42cesses (handle) 43 OUT intercomm intercommunicator between original group and the 44 newly spawned group (handle) 45OUT array_of_errcodes one code per process (array of integer) 46 47

int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,

MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,	1
<pre>int array_of_errcodes[])</pre>	2
MDI Comm appun (command argue mayaraca info root comm intercomm	$_{3}$ ticket-248T.
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,	4
array_of_errcodes, ierror) BIND(C)	5
CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)	6
INTEGER, INTENT(IN) :: maxprocs, root	7
TYPE(MPI_Info), INTENT(IN) :: info	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
TYPE(MPI_Comm), INTENT(OUT) :: intercomm	10
INTEGER :: array_of_errcodes(*)	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,	13
ARRAY_OF_ERRCODES, IERROR)	14
CHARACTER*(*) COMMAND, ARGV(*)	15
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),	16
IERROR	17
	18
{MPI::Intercomm MPI::Intracomm::Spawn(const char* command,	19
<pre>const char* argv[], int maxprocs, const MPI::Info& info,</pre>	20
<pre>int root, int array_of_errcodes[]) const(binding deprecated, see</pre>	21
Section 15.2 }	22
{MPI:::Intercomm MPI:::Intracomm::Spawn(const char* command,	23
const char* argv[], int maxprocs, const MPI::Info& info,	24
int root) const(binding deprecated, see Section 15.2) }	25
	26
MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program spec-	27
ified by command, establishing communication with them and returning an intercommun-	28
icator. The spawned processes are referred to as children. The children have their own	29
MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is	30
collective over comm, and also may not return until MPI_INIT has been called in the chil-	31
dren. Similarly, MPI_INIT in the children may not return until all parents have called	32
$MPI_COMM_SPAWN.$ In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in	33
the children form a collective operation over the union of parent and child processes. The	34
intercommunicator returned by MPI_{COMM}_{SPAWN} contains the parent processes in the	35
local group and the child processes in the remote group. The ordering of processes in the	36
local and remote groups is the same as the ordering of the group of the comm in the parents	37
and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be	38

Advice to users. An implementation may automatically establish communication before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN in the parent does not necessarily mean that MPI_INIT has been called in the children (although the returned intercommunicator can be used immediately). (End of advice to users.)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing

Unofficial Draft for Comment Only

obtained in the children through the function MPI_COMM_GET_PARENT.

39 40

 41

42

43

44

45 46

47

 48

spaces are stripped. MPI does not specify how to find the executable or how the working
 directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in \$HOME/pvm3/bin/\$PVM_ARCH). An MPI implementation running under POE on an IBM SP would use POE's method of finding executables. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (End of advice to implementors.)

If the program named in **command** does not call MPI_INIT, but instead forks a process that calls MPI_INIT, the results are undefined. Implementations may allow this case to work but are not required to.

Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

25

36

37

38

39

40

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and C++ and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI_ARGV_NULL may be used in C, C++ and Fortran to indicate an empty argument list. In C and C++, this constant is the same as NULL.

³⁴ **Example 10.1** Examples of argv in C and Fortran

³⁵ To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:

```
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

or, if not everything is known at compile time:

41	char *command;
42	char **argv;
43	<pre>command = "ocean";</pre>
44	<pre>argv=(char **)malloc(3 * sizeof(char *));</pre>
45	<pre>argv[0] = "-gridfile";</pre>
46	<pre>argv[1] = "ocean1.grd";</pre>
47	<pre>argv[2] = NULL;</pre>
48	<pre>MPI_Comm_spawn(command, argv,);</pre>

5

6

7

8

9

10

11

12

13

14 15

16

17

1

2

3

4

5

6

7

8 9

10

11

12 13

14

15

16

17

18

19

20

21

22 23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45 46 47

In Fortran:

```
CHARACTER*25 command, argv(3)
command = ' ocean '
argv(1) = ' -gridfile '
argv(2) = ' ocean1.grd'
argv(3) = ' '
call MPI_COMM_SPAWN(command, argv, ...)
```

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI_COMM_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] of MPI_COMM_SPAWN, etc. Second, argv of MPI_COMM_SPAWN must be null-terminated, so that its length can be determined. Passing an argv of MPI_ARGV_NULL to MPI_COMM_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in **argv** of **main()**, the MPI implementation may add the arguments to the **argv** that is passed to MPI_INIT.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI_ERR_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set $\{m_i : 0 \le m_i \le \text{maxprocs}\}$ of allowed values for the number of processes spawned. The set $\{m_i\}$ does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes,

MPI_COMM_SPAWN returns successfully and the number of spawned processes, *m*, is given by the size of the remote group of intercomm. If *m* is less than maxproc, reasons why the other processes were not spawned are given in array_of_errcodes as described below. If it is not possible to spawn one of the allowed numbers of processes, MPI_COMM_SPAWN raises an error of class MPI_ERR_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than maxprocs processes may be returned is called soft. See Section 10.3.4 on page 400 for more information on the soft key for info.

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N," you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0...N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the mpi_f08 module, MPI::Info in C++ and

Unofficial Draft for Comment Only

 48 ticket231-C.

- ticket231-C. 1
 - INTEGER in Fortran with the mpi module or the include file mpif.h. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated char*
 in C, character*(*) in Fortran). Routines to create and manipulate the info argument are described in Section 9 on page 383.

For the SPAWN calls, info provides additional (and possibly implementation-dependent)
 instructions to MPI and the runtime system on how to start processes. An application may
 pass MPI_INFO_NULL in C or Fortran. Portable programs not requiring detailed control over
 process locations should use MPI_INFO_NULL.

⁹ MPI does not specify the content of the info argument, except to reserve a number of ¹⁰ special key values (see Section 10.3.4 on page 400). The info argument is quite flexible and ¹¹ could even be used, for example, to specify the executable and its command-line arguments. ¹² In this case the command argument to MPI_COMM_SPAWN could be empty. The ability to ¹³ do this follows from the fact that MPI does not specify how an executable is found, and the ¹⁴ info argument can tell the runtime system where to "find" the executable "" (empty string). ¹⁵ Of course a program that does this will not be portable across MPI implementations.

16

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

20

34

35

40

42

21The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in 22which MPI reports the status of each process that MPI was requested to start. If all maxprocs 23processes were spawned, $\operatorname{array_of}$ errcodes is filled in with the value MPI_SUCCESS. If only m 24 $(0 \le m \le maxprocs)$ processes are spawned, m of the entries will contain MPI_SUCCESS and 25the rest will contain an implementation-specific error code indicating the reason MPI could 26not start the process. MPI does not specify which entries correspond to failed processes. 27An implementation may, for instance, fill in error codes in one-to-one correspondence with 28a detailed specification in the info argument. These error codes all belong to the error 29 class MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an 30 application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. In 31 C++ this constant does not exist, and the array_of_errcodes argument may be omitted from 32 the argument list. 33

Advice to implementors. MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4 on page 15. (End of advice to implementors.)

MPI_COMM_GET_PARENT(parent)

```
41 OUT parent
```

the parent communicator (handle)

```
43 int MPI_Comm_get_parent(MPI_Comm *parent)
```

```
ticket-248
T<br/>. _{\rm 44}
```

⁴⁴ MPI_Comm_get_parent(parent, ierror) BIND(C) TYPE(MPI_Comm), INTENT(OUT) :: parent INTEGER, OPTIONAL, INTENT(OUT) :: ierror

⁴⁸ MPI_COMM_GET_PARENT(PARENT, IERROR)

INTEGER PARENT, IERROR	1
<pre>{static MPI::Intercomm MPI::Comm::Get_parent()(binding deprecated, see</pre>	2 3
Section 15.2 }	4
If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE,	5
MPI_COMM_GET_PARENT returns the "parent" intercommunicator of the current process.	6
This parent intercommunicator is created implicitly inside of MPI_INIT and is the same in-	7
tercommunicator returned by SPAWN in the parents.	8
If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.	9
After the parent communicator is freed or disconnected, MPI_COMM_GET_PARENT	10
returns MPI_COMM_NULL.	11
	12
Advice to users. MPI_COMM_GET_PARENT returns a handle to a single intercom-	13 14
municator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI_COMM_DISCONNECT or	14
MPI_COMM_FREE will cause other references to the intercommunicator to become	16
invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator	17
is not useful. (End of advice to users.)	18
	19
Rationale. The desire of the Forum was to create a constant	20
MPI_COMM_PARENT similar to MPI_COMM_WORLD. Unfortunately such a constant	21
cannot be used (syntactically) as an argument to MPI_COMM_DISCONNECT, which	22
is explicitly allowed. (End of rationale.)	23 24
10.3.3 Starting Multiple Executables and Establishing Communication	24 25
	26
While MPI_COMM_SPAWN is sufficient for most cases, it does not allow the spawning	27
of multiple binaries, or of the same binary with multiple sets of arguments. The follow-	28
ing routine spawns multiple binaries or the same binary with multiple sets of arguments,	29
establishing communication with them and placing them in the same MPI_COMM_WORLD.	30
	31
	32
	$33 \\ 34$
	35
	36
	37
	38
	39
	40
	41
	42 43
	43 44
	45
	46
	47

1

 $\mathbf{2}$

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes)

```
3
                   IN
                             count
                                                         number of commands (positive integer, significant to
            4
                                                         MPI only at root — see advice to users)
            5
                   IN
                             array_of_commands
                                                         programs to be executed (array of strings, significant
            6
                                                         only at root)
            7
            8
                   IN
                                                         arguments for commands (array of array of strings,
                             array_of_argv
            9
                                                         significant only at root)
            10
                   IN
                             array_of_maxprocs
                                                         maximum number of processes to start for each com-
            11
                                                         mand (array of integer, significant only at root)
           12
                   IN
                             array_of_info
                                                         info objects telling the runtime system where and how
           13
                                                         to start processes (array of handles, significant only at
           14
                                                         root)
            15
            16
                   IN
                                                         rank of process in which previous arguments are ex-
                             root
            17
                                                         amined (integer)
            18
                   IN
                                                         intracommunicator containing group of spawning pro-
                             comm
            19
                                                         cesses (handle)
           20
                   OUT
                                                         intercommunicator between original group and newly
                             intercomm
           21
                                                         spawned group (handle)
           22
           23
                   OUT
                             array_of_errcodes
                                                         one error code per process (array of integer)
           ^{24}
           25
                 int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
           26
  ticket140.
                                 char **array_of_argv[], const int array_of_maxprocs[], const
  ticket140.<sup>27</sup>
                                MPI_Info array_of_info[], int root, MPI_Comm comm,
            28
                                MPI_Comm *intercomm, int array_of_errcodes[])
ticket-248T. 29
                 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
           30
                                array_of_maxprocs, array_of_info, root, comm, intercomm,
           ^{31}
                                array_of_errcodes, ierror) BIND(C)
           32
                      INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
           33
                      CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
           34
                      array_of_argv(count, *)
           35
                      TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
           36
                      TYPE(MPI_Comm), INTENT(IN) :: comm
           37
                      TYPE(MPI_Comm), INTENT(OUT) :: intercomm
           38
                      INTEGER :: array_of_errcodes(*)
           39
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            40
           41
                 MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
           42
                                ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
           43
                                ARRAY_OF_ERRCODES, IERROR)
           44
                      INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
            45
                      INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
            46
                      CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
            47
            48
```

```
{MPI:::Intercomm MPI:::Intracomm::Spawn_multiple(int count,
                                                                                      \mathbf{2}
              const char* array_of_commands[], const char** array_of_argv[],
              const int array_of_maxprocs[],
                                                                                      4
              const MPI::Info array_of_info[], int root,
                                                                                      5
              int array_of_errcodes[] (binding deprecated, see Section 15.2) }
                                                                                      6
{MPI:::Intercomm MPI::Intracomm::Spawn_multiple(int count,
                                                                                      7
              const char* array_of_commands[], const char** array_of_argv[],
              const int array_of_maxprocs[],
                                                                                      9
              const MPI::Info array_of_info[], int root) (binding deprecated, see
                                                                                      10
              Section 15.2 }
                                                                                      11
```

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the j-th argument to command number i.

This may seem backwards to Fortran programmers who are familiar Rationale. with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv must be the same as count. (*End of rationale.*)

Advice to users. The argument count is interpreted by MPI only at the root, as is array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array_of_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (End of advice to users.)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be (char *******)0 in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first element is null ((char *)0 in C and empty string in Fortran). In Fortran at non-root processes, the count argument must be set to a value that is consistent with the provided array_of_argv although the content of these arguments has no meaning for this operation.

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the commands are specified in MPI_COMM_SPAWN_MULTIPLE. Assume that m_1 processes are generated by the first command, m_2 by the second, etc. The processes corresponding to the first command have ranks $0, 1, \ldots, m_1 - 1$. The processes in the second command have ranks $m_1, m_1 + 1, \ldots, m_1 + 1$ $m_2 - 1$. The processes in the third have ranks $m_1 + m_2, m_1 + m_2 + 1, \dots, m_1 + m_2 + m_3 - 1$, etc.

Calling MPI_COMM_SPAWN multiple times would create many Advice to users. sets of children with different MPI_COMM_WORLDs whereas MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD,

Unofficial Draft for Comment Only

ticket229.1. 35

1

3

8

12 13

14

15

16

17

18

19

20

2122

23

 24

25

26

27

2829

30

31

32

33

34

36

37

38

39

40

41

42

43

4445

46

47

so the two methods are not completely equivalent. There are also two performancerelated reasons why, if you need to spawn multiple executables, you may want to use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (*End of advice to users.*)

⁹ The array_of_errcodes argument is a 1-dimensional array of size $\sum_{i=1}^{count} n_i$, where n_i is ¹⁰ the *i*-th element of array_of_maxprocs. Command number *i* corresponds to the n_i contiguous ¹¹ slots in this array from element $\sum_{j=1}^{i-1} n_j$ to $\left[\sum_{j=1}^{i} n_j\right] - 1$. Error codes are treated as for ¹² MPI_COMM_SPAWN.

```
Example 10.2 Examples of array_of_argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program
"atmos" with argument "atmos.grd" in C:
char *array_of_commands[2] = {"ocean", "atmos"};
char *array_of_argv[2];
char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
```

```
char *argv1[] = {"atmos.grd", (char *)0};
```

```
array_of_argv[0] = argv0;
array_of_argv[1] = argv1;
```

```
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
```

Here's how you do it in Fortran:

```
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = ' ocean '
array_of_argv(1, 1) = ' -gridfile '
array_of_argv(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '
commands(2) = ' atmos '
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '
```

```
call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
```

```
<sup>39</sup> 10.3.4 Reserved Keys
```

- The following keys are reserved. An implementation is not required to interpret these keys, but if it does interpret the key, it must provide the functionality described.
- $_{44}^{43}$ host Value is a hostname. The format of the hostname is determined by the implementation.
- ⁴⁵ arch Value is an architecture name. Valid architecture names and what they mean are
 ⁴⁶ determined by the implementation.
- 47 48

1

 $\mathbf{2}$

3

4

5

6

7

8

21

22

23

24 25

26

27

28

29

30

31 32

33 34

35 36

- wdir Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.
- path Value is a directory or set of directories where the implementation should look for the executable. The format of path is determined by the implementation.
- file Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.
- soft Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a comma-separated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

- 1. a means a
- 2. a:b means a, a + 1, a + 2, ..., b
- 3. a:b:c means $a, a + c, a + 2c, \ldots, a + ck$, where for c > 0, k is the largest integer for which $a + ck \le b$ and for c < 0, k is the largest integer for which $a + ck \ge b$. If b > a then c must be positive. If b < a then c must be negative.

```
Examples:
```

1. a:b gives a range between a and	10)
------------------------------------	----	---

- 2. O:N gives full "soft" functionality
- 3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows power-of-two number of processes.
- 4. 2:10000:2 allows even number of processes.
- 5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.

10.3.5 Spawn Example

Manager-worker Example [,] Using MPI_COMM_SPAWN.

 $\frac{24}{25}$

 $45 \\ 46$

ticket0.

```
1
        if (world_size != 1)
                                  error("Top heavy with management");
2
3
        MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
4
                           &universe_sizep, &flag);
5
        if (!flag) {
6
             printf("This MPI does not support UNIVERSE_SIZE. How many\n\
7
     processes total?");
8
             scanf("%d", &universe_size);
9
        } else universe_size = *universe_sizep;
10
        if (universe_size == 1) error("No room to start workers");
11
        /*
12
13
         * Now spawn the workers. Note that there is a run-time determination
14
         * of what type of worker to spawn, and presumably this calculation must
15
         * be done at run time and cannot be calculated before starting
16
         * the program. If everything is known when the application is
17
         * first started, it is generally better to start them all at once
18
         * in a single MPI_COMM_WORLD.
19
         */
20
21
        choose_worker_program(worker_program);
22
        MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
23
                  MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
24
                  MPI_ERRCODES_IGNORE);
25
        /*
26
         * Parallel code here. The communicator "everyone" can be used
27
         * to communicate with the spawned processes, which have ranks 0,...
28
         * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
29
         * "everyone".
30
         */
31
32
        MPI_Finalize();
33
        return 0;
34
     }
35
     /* worker */
36
37
38
     #include "mpi.h"
     int main(int argc, char *argv[])
39
40
     ſ
41
        int size;
42
        MPI_Comm parent;
        MPI_Init(&argc, &argv);
43
        MPI_Comm_get_parent(&parent);
44
        if (parent == MPI_COMM_NULL) error("No parent!");
45
        MPI_Comm_remote_size(parent, &size);
46
47
        if (size != 1) error("Something's wrong with the parent");
48
```

```
1
   /*
                                                                                      2
    * Parallel code here.
                                                                                      3
    * The manager is represented as the process with rank 0 in (the remote
    * group of) the parent communicator. If the workers need to communicate
                                                                                      4
    * among themselves, they can use MPI_COMM_WORLD.
                                                                                      5
                                                                                      6
    */
                                                                                      7
                                                                                      8
   MPI_Finalize();
                                                                                      9
   return 0;
}
                                                                                      10
                                                                                      11
                                                                                      12
                                                                                      13
                                                                                      14
10.4
       Establishing Communication
                                                                                      15
                                                                                      16
This section provides functions that establish communication between two sets of MPI
                                                                                      17
```

processes that do not share a communicator.

Some situations in which these functions are useful are:

- 1. Two parts of an application that are started independently need to communicate.
- 2. A visualization tool wants to attach to a running process.
- 3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI intercommunicator, where the two groups of the intercommunicator are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) server, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that doesn't participate in a collective operation may cause a server to crash or hang. (End of advice to users.)

10.4.1 Names, Addresses, Ports, and All That

Almost all of the complexity in MPI client/server routines addresses the question "how does the client find out how to contact the server?" The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication.

Unofficial Draft for Comment Only

18

19 20

21

22 23

 24

2526

27

28

2930

 31

32 33

34

35

36

37

38

39

40

41 42

43 44

45

46

47

1 2 3 4 5 6 7	Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client doesn't really care what server it contacts, only that it be able to get in touch with one that can handle its request. Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple portable code. The following should be compatible with MPI:
8 9 10 11	The server resides at a well-known internet address host:port.The server prints out an address to the terminal, the user gives this address to the client program.
12 13 14	• The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
15 16 17	• The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.
 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 	MPI does not require a nameserver, so not all implementations will be able to support all of the above scenarios. However, MPI provides an optional nameserver interface, and is compatible with external name servers. A port_name is a <i>system-supplied</i> string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a port_name with the MPI_OPEN_PORT routine. It accepts a connection to a given port with MPI_COMM_ACCEPT. A client uses port_name to connect to the server. By itself, the port_name mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate port_name to the client. It would be more convenient if a server could specify that it be known by an <i>application-supplied</i> service_name. An MPI implementation may allow the server to publish a (port_name, service_name) pair with MPI_PUBLISH_NAME and the client to retrieve the port name from the service name with MPI_LOOKUP_NAME. This allows three levels of portability, with increasing levels of functionality.
34 35 36	1. Applications that do not rely on the ability to publish names are the most portable. Typically the port_name must be transferred "by hand" from server to client.
37 38 39 40 41	2. Applications that use the MPI_PUBLISH_NAME mechanism are completely portable among implementations that provide this service. To be portable among all imple- mentations, these applications should have a fall-back mechanism that can be used when names are not published.
41 42 43 44 45 46 47 48	3. Applications may ignore MPI's name publishing functionality and use their own mech- anism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.

10.4.2	Server Routines		1		
A server	makes itself available with	n two routines. First it must call MPI_OPEN_PORT to	2		
		contacted. Secondly it must call MPI_COMM_ACCEPT	3		
	connections from clients.	·	4 5		
-			6		
			7		
MPI_OPI	EN_PORT(info, port_name)	8		
IN	info	implementation-specific information on how to estab-	9		
		lish an address (handle)	10		
OUT	port_name	newly established port (string)	11		
			12		
int MPI	_Open_port(MPI_Info in:	fo, char *port_name)	13		
MDT Open	n_port(info, port_name	ierror) PIND(C)	$_{14}$ ticket-248T		
	E(MPI_Info), INTENT(IN		15		
		T_NAME), INTENT(OUT) :: port_name	16		
	EGER, OPTIONAL, INTENT	•	17 18		
			19		
	N_PORT(INFO, PORT_NAME	, IERROR)	20		
	RACTER*(*) PORT_NAME EGER INFO, IERROR		21		
	LGER INFU, IERRUR		22		
{void MH		I::Info& info, char* port_name)(binding	23		
	deprecated, see Sec	tion 15.2 }	24		
This	function establishes a net	work address, encoded in the port_name string, at which	25		
the server will be able to accept connections from clients. port_name is supplied by the					
	possibly using information		27		
MPI	copies a system-supplied p	ort name into port_name. port_name identifies the newly	28		
		client to contact the server. The maximum size string	29		
that may	be supplied by the system	n is MPI_MAX_PORT_NAME.	30 31		
Ad	vice to users. The system	copies the port name into port_name. The application	32		
	-	at size to hold this value. (End of advice to users.)	33		
port	name is essentially a not	work address. It is unique within the communication	34		
	c c	nined by the implementation), and may be used by any	35		
	- ,	universe. For instance, if it is an internet (host:port)	36		
		ternet. If it is a low level switch address on an IBM SP,	37		
	unique to that SP.	,	38		
Ad	vice to implementors. T	hese examples are not meant to constrain implementa-	39		
	*	r instance, contain a user name or the name of a batch	40		
	•	within some well-defined communication domain. The	41		
-		main, the more useful MPI's client/server functionality	42 43		
-	be. (End of advice to imp		43		
		aplementation-defined. For instance, an internet address	45		
-		, or anything that the implementation can decode into	46		
-		be reused after it is freed with MPI_CLOSE_PORT and	47		
	by the system.		48		

405

	1 2 3 4	to ch	-	he user may type in port_name by hand, it is useful dable and does not have embedded spaces. (<i>End of</i>		
	5 6 7			nentation how to establish the address. It may, and r to get the implementation defaults.		
	8 9	MPI CLOS	E_PORT(port_name)			
	9 10	IN	port_name	a port (string)		
	11					
ticket140 ticket-248T		int MPI_C	lose_port(<mark>const</mark> char *po	rt_name)		
ticket-2401	14 15 16	<pre>MPI_Close_port(port_name, ierror) BIND(C) CHARACTER(LEN=*), INTENT(IN) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
	17 18 19 20	CHARA	_PORT(PORT_NAME, IERROR) CTER*(*) PORT_NAME ER IERROR			
	20 21 22	{void MPI	::Close_port(const char* }	<pre>port_name)(binding deprecated, see Section 15.2)</pre>		
	23 24	This funct	ion releases the network addr	ess represented by port_name.		
	25 26	MPI_COM	M_ACCEPT(port_name, info,	root, comm, newcomm)		
	27	IN	port_name	port name (string, used only on root)		
	28 29 30	IN	info	implementation-dependent information (handle, used only on <code>root</code>)		
	31	IN	root	rank in comm of root node (integer)		
	32 33 34	IN	comm	intracommunicator over which call is collective (handle)		
	35 36	OUT	newcomm	intercommunicator with client as remote group (handle)		
ticket140	37	int MDT C	omm accont(const char *n	ort_name, MPI_Info info, int root,		
	39	IIIC III 1_0	MPI_Comm comm, MPI_C			
ticket-248T	• 40 41 42	<pre>MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) BIND(C) CHARACTER(LEN=*), INTENT(IN) :: port_name</pre>				
	42	TYPE(MPI_Info), INTENT(IN) :: info				
	44	INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: comm				
	45	TYPE(MPI_Comm), INTENT(IN) comm TYPE(MPI_Comm), INTENT(OUT) :: newcomm				
	46 47	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
	48	MPI_COMM_	ACCEPT(PORT_NAME, INFO,	ROOT, COMM, NEWCOMM, IERROR)		

CHAPTER 10. PROCESS CREATION AND MANAGEMENT

406

			1		
	ACTER*(*) PORT_NAME ER INFO, ROOT, COMM, NEW(COMM, IERROR	2		
	3				
$\{MPI::Int$	4				
	5				
	Section 15.2 }		6		
MPI_0	7				
-	nmunicator. It returns an inter	communicator that allows communication with the	8		
client.		blicked through a call to MDL ODEN DODT	10		
•		ablished through a call to MPI_OPEN_PORT. ring that may allow fine control over the ACCEPT	11		
call.	a implementation-defined str	ing that may allow line control over the ACCEPT	12		
can.			13		
10.4.3 C	lient Routines		14		
10.4.5 C	nent Routines		15		
There is or	nly one routine on the client s	ide.	16		
			17		
MPL COM	M_CONNECT(port_name, info	o root comm newcomm)	18		
		,	19		
IN	port_name	network address (string, used only on root)	20		
IN	info	implementation-dependent information (handle, used	21		
		only on root)	22		
IN	root	rank in comm of root node (integer)	23 24		
IN	comm	intracommunicator over which call is collective (han-	24		
		dle)	26		
OUT	newcomm	intercommunicator with server as remote group (han-	27		
001	dle)				
			29		
int MPT (comm connect(const char *	port_name, MPI_Info info, int root,	30 ticket 140.		
110 111 1_0	31				
	MPI_Comm comm, MPI_C		32 ticket-248T.		
		root, comm, newcomm, ierror) BIND(C)	33		
	ACTER(LEN=*), INTENT(IN)	-	34		
	<pre>(MPI_Info), INTENT(IN) ::</pre>	1110	35		
	ER, INTENT(IN) :: root (MPI_Comm), INTENT(IN) ::	comm	36 37		
	(MPI_Comm), INTENT(IN)		38		
	ER, OPTIONAL, INTENT(OUT)		39		
			40		
		ROOT, COMM, NEWCOMM, IERROR)	41		
	ACTER*(*) PORT_NAME		42		
INTEG	ER INFO, ROOT, COMM, NEW	JUMM, LEKKUK	43		
$\{\texttt{MPI}::\texttt{Int}$	ercomm MPI:::Intracomm::Co	onnect(const char* port_name,	44		
		o, int root) const(binding deprecated, see	45		
	Section 15.2 }		46		
			47		
			48		

¹ This routine establishes communication with a server specified by port_name. It is ² collective over the calling communicator and returns an intercommunicator in which the ³ remote group participated in an MPI_COMM_ACCEPT.

If the named port does not exist (or has been closed), MPI_COMM_CONNECT raises
 an error of class MPI_ERR_PORT.

If the port exists, but does not have a pending MPI_COMM_ACCEPT, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls MPI_COMM_ACCEPT. In the case of a time out, MPI_COMM_CONNECT
 raises an error of class MPI_ERR_PORT.

10 11

12

13

14

15

Advice to implementors. The time out period may be arbitrarily short or long. However, a high quality implementation will try to queue connection attempts so that a server can handle simultaneous requests from several clients. A high quality implementation may also provide a mechanism, through the info arguments to MPI_OPEN_PORT, MPI_COMM_ACCEPT and/or MPI_COMM_CONNECT, for the user to control timeout and queuing behavior. (*End of advice to implementors.*)

16 17 18

19

20

21

22

23

 24

25 26

27 28

29

30

 31

32

33

34

35

36 37 38

39

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

port_name is the address of the server. It must be the same as the name returned by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port_name, an implementation may accept them as well. For instance, if port_name is (hostname:port), an implementation may accept (ip_address:port) as well.

10.4.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A (service_name, port_name) pair is published by the server, and may be retrieved by a client using the service_name only. An MPI implementation defines the *scope* of the service_name, that is, the domain over which the service_name can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. Implementations should document how the scope is determined. High-quality implementations will give some control to users through the info arguments to name publishing functions. Examples are given in the descriptions of individual functions.

```
MPI_PUBLISH_NAME(service_name, info, port_name)
```

40	, IN	service_name	a service name to associate with the port (string)
41	IN	info	implementation-specific information (handle)
42	² IN	port_name	a port name (string)
43	3		
${{ m ticket140.}\atop{ m ticket140.}}^{44}$	5	MPI_Publish_name(<mark>cons</mark> char *port_n	t char *service_name, MPI_Info info, <mark>const</mark> ame)
47		Publish_name(service_	name, info, port_name, ierror) BIND(C)
48	3	TYPE(MPI_Info), INTEN	T(IN) :: info

```
1
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
                                                                                    2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    3
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
                                                                                    4
    INTEGER INFO, IERROR
                                                                                    5
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
                                                                                    6
                                                                                    7
{void MPI::Publish_name(const char* service_name, const MPI::Info& info,
                                                                                    8
              const char* port_name) (binding deprecated, see Section 15.2) }
                                                                                    9
```

This routine publishes the pair (port_name, service_name) so that an application may retrieve a system-supplied port_name using a well-known service_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI_PUBLISH_NAME.

MPI permits publishing more than one service_name for a single port_name. On the other hand, if service_name has already been published within the scope determined by info, the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by MPI_LOOKUP_NAME.

Note that while service_name has a limited scope, determined by the implementation, port_name always has global scope within the communication universe used by the implementation (i.e., it is globally unique).

port_name should be the name of a port established by MPI_OPEN_PORT and not yet deleted by MPI_CLOSE_PORT. If it is not, the result is undefined.

Advice to implementors. In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of "ocean" running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to MPI_PUBLISH_NAME after the first may fail. There is no universal solution to this.

To handle these situations, a high-quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

 24

1 MPI_UNPUBLISH_NAME(service_name, info, port_name) 2 IN service_name a service name (string) 3 IN info implementation-specific information (handle) 4 5IN port_name a port name (string) 6 ticket140. int MPI_Unpublish_name(const char *service_name, MPI_Info info, const ticket140. char *port_name) 9 ticket-248T. MPI_Unpublish_name(service_name, info, port_name, ierror) BIND(C) 10 CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name 11 TYPE(MPI_Info), INTENT(IN) :: info 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) 15INTEGER INFO, IERROR 16CHARACTER*(*) SERVICE_NAME, PORT_NAME 17{void MPI::Unpublish_name(const char* service_name, const MPI::Info& info, 18 const char* port_name) (binding deprecated, see Section 15.2) 19 20This routine unpublishes a service name that has been previously published. Attempt-21ing to unpublish a name that has not been published or has already been unpublished is 22erroneous and is indicated by the error class MPI_ERR_SERVICE. 23All published names must be unpublished before the corresponding port is closed and 24 before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implemen-25tation dependent when a process tries to unpublish a name that it did not publish. 26If the info argument was used with MPI_PUBLISH_NAME to tell the implementation 27how to publish names, the implementation may require that info passed to 28MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish 29 a name. 30 31 32 MPI_LOOKUP_NAME(service_name, info, port_name) 33 IN service_name a service name (string) 34 IN info implementation-specific information (handle) 35 36 OUT port_name a port name (string) 37 38 ticket140. int MPI_Lookup_name(const char *service_name, MPI_Info info, 39 char *port_name) ticket-248T. 40 MPI_Lookup_name(service_name, info, port_name, ierror) BIND(C) 41 CHARACTER(LEN=*), INTENT(IN) :: service_name 42TYPE(MPI_Info), INTENT(IN) :: info 43 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) 47CHARACTER*(*) SERVICE_NAME, PORT_NAME 48

INTEGER INFO, IERROR	1
<pre>{void MPI::Lookup_name(const char* service_name, const MPI::Info& info,</pre>	2 3 4
This function retrieves a port_name published by MPI_PUBLISH_NAME with service_name. If service_name has not been published, it raises an error in the error class MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the largest possible port name (see discussion above under MPI_OPEN_PORT). If an implementation allows multiple entries with the same service_name within the same scope, a particular port_name is chosen in a way determined by the implementation. If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.	5 6 7 8 9 10 11 12 13
10.4.5 Reserved Key Values	14
The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.	15 16 17
<pre>ip_port Value contains IP port number at which to establish a port. (Reserved for MPI_OPEN_PORT only).</pre>	18 19
ip_address Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI_OPEN_PORT call is made, the results are undefined. (Reserved for MPI_OPEN_PORT only).	20 21 22 23
10.4.6 Client/Server Examples	24 25
Simplest Example — Completely Portable.	26
The following example shows the simplest way to use the client/server interface. It does not use service names at all. On the server side:	27 28 29 30
<pre>char myport[MPI_MAX_PORT_NAME];</pre>	31 32
MPI_Comm intercomm;	33
/* */	34 35
MPI_Open_port(MPI_INFO_NULL, myport);	36
<pre>printf("port name is: %s\n", myport);</pre>	37
<pre>MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm); /* do something with intercomm */</pre>	38 39
The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports stdin such that this works). On the client side:	40 41 42 43
<pre>MPI_Comm intercomm; char name[MPI_MAX_PORT_NAME]; printf("enter port name: "); gets(name); MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);</pre>	44 45 46 47 48
In I_COMM_CONNECCURAME, IN I_INIU_NOLL, C, IN I_COUNT_DELL, &INUELCOUND),	

```
1
     Ocean/Atmosphere - Relies on Name Publishing
\mathbf{2}
     In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere
3
     climate model. It assumes that the MPI implementation publishes names.
4
5
6
          MPI_Open_port(MPI_INFO_NULL, port_name);
7
          MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
8
9
          MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
10
          /* do something with intercomm */
11
          MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
12
13
14
     On the client side:
15
          MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
16
          MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
17
                              &intercomm);
18
19
     Simple Client-Server Example.
20
21
     This is a simple example; the server accepts only a single connection at a time and serves
22
     that connection until the client requests to be disconnected. The server is a single process.
23
         Here is the server. It accepts a single connection and then processes data until it
^{24}
     receives a message with tag 1. A message with tag 0 tells the server to exit.
25
26
     #include "mpi.h"
27
     int main( int argc, char **argv )
28
     {
29
          MPI_Comm client;
30
          MPI_Status status;
31
          char port_name[MPI_MAX_PORT_NAME];
32
          double buf[MAX_DATA];
33
          int
                  size, again;
34
35
          MPI_Init( &argc, &argv );
36
          MPI_Comm_size(MPI_COMM_WORLD, &size);
37
          if (size != 1) error(FATAL, "Server too big");
38
          MPI_Open_port(MPI_INFO_NULL, port_name);
39
          printf("server available at %s\n",port_name);
40
          while (1) {
41
              MPI_Comm_accept( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
42
                                 &client );
43
              again = 1;
44
              while (again) {
45
                  MPI_Recv( buf, MAX_DATA, MPI_DOUBLE,
46
                              MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status );
47
                   switch (status.MPI_TAG) {
48
                       case 0: MPI_Comm_free( &client );
```

```
1
                           MPI_Close_port(port_name);
                                                                                         \mathbf{2}
                          MPI_Finalize();
                                                                                         3
                           return 0;
                  case 1: MPI_Comm_disconnect( &client );
                                                                                         4
                           again = 0;
                                                                                         5
                                                                                         6
                           break;
                                                                                         7
                  case 2: /* do something */
                                                                                         8
                  . . .
                                                                                         9
                  default:
                                                                                         10
                           /* Unexpected message type */
                                                                                         11
                           MPI_Abort( MPI_COMM_WORLD, 1 );
                  }
                                                                                         12
             }
                                                                                         13
        }
                                                                                         14
                                                                                         15
}
                                                                                         16
    Here is the client.
                                                                                         17
                                                                                         18
#include "mpi.h"
                                                                                         19
int main( int argc, char **argv )
                                                                                         20
{
                                                                                         21
    MPI_Comm server;
                                                                                         22
    double buf [MAX_DATA];
                                                                                         23
    char port_name[MPI_MAX_PORT_NAME];
                                                                                         ^{24}
                                                                                         25
    MPI_Init( &argc, &argv );
                                                                                         26
    strcpy(port_name, argv[1] );/* assume server's name is cmd-line arg */
                                                                                         27
                                                                                         28
    MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                                                                                         29
                        &server );
                                                                                         30
                                                                                         ^{31}
    while (!done) {
                                                                                         32
        tag = 2; /* Action to perform */
                                                                                         33
        MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
                                                                                         34
        /* etc */
                                                                                         35
        7
                                                                                         36
    MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
                                                                                         37
    MPI_Comm_disconnect( &server );
                                                                                         38
    MPI_Finalize();
                                                                                         39
    return 0;
                                                                                         40
}
                                                                                         41
                                                                                         42
                                                                                         43
```

10.5 Other Functionality

10.5.1 Universe Size

Many "dynamic" MPI applications are expected to exist in a static runtime environment, in which resources have been allocated before the application is run. When a user (or

Unofficial Draft for Comment Only

44

 $45 \\ 46$

47

possibly a batch system) runs one of these quasi-static applications, she will usually specify
 a number of processes to start and a total number of processes that are expected. An
 application simply needs to know how many slots there are, i.e., how many processes it
 should spawn.

5MPI provides an attribute on MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, that allows 6 the application to obtain this information in a portable manner. This attribute indicates 7the total number of processes that are expected. In Fortran, the attribute is the integer 8 value. In C, the attribute is a pointer to the integer value. An application typically subtracts 9 the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it 10 should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If 11defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE 12is determined by the application startup mechanism in a way not specified by MPI. (The 13size of MPI_COMM_WORLD is another example of such a parameter.)

- Possibilities for how MPI_UNIVERSE_SIZE might be set include
- A -universe_size argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set.

MPI_UNIVERSE_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

³⁰ MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, ³¹ and is in essence a portable mechanism to allow the user to pass to the application (through ³² the MPI process startup mechanism, such as mpiexec) a piece of critical runtime informa-³³ tion. Note that no interaction with the runtime environment is required. If the runtime ³⁴ environment changes size while an application is running, MPI_UNIVERSE_SIZE is not up-³⁵ dated, and the application must find out about the change through direct communication ³⁶ with the runtime system.

37 38

39

14

15

16 17

18

19 20

21

22 23

 24

25

26

27

28

29

10.5.2 Singleton MPI_INIT

⁴⁰ A high-quality implementation will allow any process (including those not started with a ⁴¹ "parallel application" mechanism) to become an MPI process by calling MPI_INIT. Such ⁴² a process can then connect to other MPI processes using the MPI_COMM_ACCEPT and ⁴³ MPI_COMM_CONNECT routines, or spawn other MPI processes. MPI does not mandate ⁴⁴ this behavior, but strongly encourages it where technically feasible.

- $45 \\ 46$
- Advice to implementors. To start MPI processes belonging to the same
- 47 MPI_COMM_WORLD requires some special coordination. The processes must be started 48 at the "same" time, they must have a mechanism to establish communication, etc.

Either the user or the operating system must take special steps beyond simply starting processes.

When an application enters MPI_INIT, clearly it must be able to determine if these special steps were taken. If a process enters MPI_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI_COMM_WORLD has size 1.

In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either

- 1. Create the environment (e.g., start a daemon) or
- 2. Raise an error if it cannot create the environment and the environment has not been started independently.

A high-quality implementation will try to create a singleton MPI process and not raise an error.

(End of advice to implementors.)

10.5.3 MPI_APPNUM

There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number that generated the current process. Numbering starts from zero. If a process was spawned with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementationspecific startup mechanism that can handle multiple process specifications, MPI_APPNUM should be set to the number of the corresponding process specification. In particular, if it is started with

mpiexec spec0 [: spec1 : spec2 : ...]

MPI_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI_COMM_SPAWN or

MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM doesn't make sense in the context of the implementation-specific startup mechanism, MPI_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

appnum Value contains an integer that overrides the default value for MPI_APPNUM in the child.

Rationale.When a single application is started, it is able to figure out how many pro-45cesses there are by looking at the size of MPI_COMM_WORLD. An application consisting46of multiple SPMD sub-applications has no way to find out how many sub-applications47there are and to which sub-application the process belongs. While there are ways to48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7

8

9

10

11

12 13

14

15

16 17

18

19

20 21

22 23

 24

25

26

27

28

29

30

31 32 33

34

35

36

37

38

39

40

41 42

43

1 figure it out in special cases, there is no general mechanism. MPI_APPNUM provides $\mathbf{2}$ such a general mechanism. (End of rationale.) 3 4 Releasing Connections 10.5.4 5Before a client and server connect, they are independent MPI applications. An error in one 6 does not affect the other. After establishing a connection with MPI_COMM_CONNECT and 7 MPI_COMM_ACCEPT, an error in one may affect the other. It is desirable for a client and 8 server to be able to disconnect, so that an error in one will not affect the other. Similarly, 9 it might be desirable for a parent and child to disconnect, so that errors in the child do not 10 affect the parent, or vice-versa. 11 12• Two processes are **connected** if there is a communication path (direct or indirect) 13 between them. More precisely: 14151. Two processes are connected if 16(a) they both belong to the same communicator (inter- or intra-, including 17 MPI_COMM_WORLD) or 18 (b) they have previously belonged to a communicator that was freed with 19 MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or 20(c) they both belong to the group of the same window or filehandle. 212. If A is connected to B and B to C, then A is connected to C. 22 23• Two processes are **disconnected** (also **independent**) if they are not connected. 2425• By the above definitions, connectivity is a transitive property, and divides the uni-26verse of MPI processes into disconnected (independent) sets (equivalence classes) of 27processes. 28• Processes which are connected, but don't share the same MPI_COMM_WORLD may be-29 come disconnected (independent) if the communication path between them is broken 30 by using MPI_COMM_DISCONNECT. 31 32 The following additional rules apply to MPI routines in other chapters: 33 34 • MPI_FINALIZE is collective over a set of connected processes. 35• MPI_ABORT does not abort independent processes. It may abort all processes in 36 the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may 37 abort connected processes as well, though it makes a "best attempt" to abort only 38 the processes in comm. 39 40 • If a process terminates without calling MPI_FINALIZE, independent processes are not 41 affected but the effect on connected processes is not defined. 4243 44MPI_COMM_DISCONNECT(comm) 45 INOUT communicator (handle) 46comm 4748 int MPI_Comm_disconnect(MPI_Comm *comm)

ticket-248T.

			1
MDT Comm	n_disconnect(comm, id	arror) BIND(C)	1
	E(MPI_Comm), INTENT()		3
	EGER, OPTIONAL, INTE		4
			5
	1_DISCONNECT(COMM, I EGER COMM, IERROR	LRRUR)	6
			7
{void MF	PI::Comm::Disconnect	() (binding deprecated, see Section 15.2) }	8
This	function waits for all	pending communication on comm to complete internally,	9 10
		bject, and sets the handle to MPI_COMM_NULL. It is a	10
collective	e operation.		12
	0	e communicator MPI_COMM_WORLD or MPI_COMM_SELF.	13
		may be called only if all communication is complete and	14
		an be delivered to its destination. This requirement is the	15
	for MPI_FINALIZE.	has the same action as MDL COMM FREE amount that it	16
		has the same action as MPI_COMM_FREE, except that it to finish internally and enables the guarantee about the	17
	of disconnected process		18
001147101	of disconnected process		19
Adt	vice to users. To disc	connect two processes you may need to call	20
MP	I_COMM_DISCONNECT	$\Gamma, \ MPI_WIN_FREE \ \mathrm{and} \ MPI_FILE_CLOSE \ \mathrm{to} \ \mathrm{remove} \ \mathrm{all}$	21 22
	-	veen the two processes. Notes that it may be necessary	22
		nunicators (or to free several windows or files) before two	24
2220	aggog are completely in		
pro	cesses are completely in	dependent. (End of advice to users.)	25
_			25 26
Rat	tionale. It would be ni	ce to be able to use MPI_COMM_FREE instead, but that	
Rat fun	tionale. It would be ni		26
Rat fun	<i>tionale.</i> It would be ni ction explicitly does no	ce to be able to use MPI_COMM_FREE instead, but that	26 27 28 29
Rat fun rate	<i>tionale.</i> It would be ni ction explicitly does no <i>tionale.</i>)	ce to be able to use MPI_COMM_FREE instead, but that	26 27 28 29 30
Rat fun rate	<i>tionale.</i> It would be ni ction explicitly does no <i>tionale.</i>)	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i>	26 27 28 29 30 31
Rat fun rate	<i>tionale.</i> It would be ni ction explicitly does no <i>tionale.</i>)	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i>	26 27 28 29 30 31 32
Rat fun rata 10.5.5	<i>tionale.</i> It would be ni ction explicitly does no <i>tionale.</i>)	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i> sh MPI Communication	26 27 28 29 30 31 32 33
Rat fun rata 10.5.5	tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i> sh MPI Communication	26 27 28 29 30 31 32
Rat fun rata 10.5.5 MPI_CON IN	<i>tionale.</i> It would be ni ction explicitly does no <i>ionale.</i>) Another Way to Establi MM_JOIN(fd, intercomm fd	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i> sh MPI Communication) socket file descriptor	26 27 28 29 30 31 32 33 34
Rat fun rata 10.5.5	tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i> sh MPI Communication	26 27 28 29 30 31 32 33 34 35
Rat fun rata 10.5.5 MPI_CON IN OUT	tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i> sh MPI Communication) socket file descriptor new intercommunicator (handle)	26 27 28 29 30 31 32 33 34 35 36
Rat fun rata 10.5.5 MPI_CON IN OUT	<i>tionale.</i> It would be ni ction explicitly does no <i>ionale.</i>) Another Way to Establi MM_JOIN(fd, intercomm fd	ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (<i>End of</i> sh MPI Communication) socket file descriptor new intercommunicator (handle)	26 27 28 29 30 31 32 33 34 35 36 37
Rat fun rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T.
Rat fun rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm INTE	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm EGER, INTENT(IN) ::</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C) fd</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T. 40
Rat fun rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm INTE TYPE	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm EGER, INTENT(IN) :: E(MPI_Comm), INTENT(I)</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C) fd DUT) :: intercomm</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T. 40 41
Rat fun rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm INTE TYPE	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm EGER, INTENT(IN) ::</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C) fd DUT) :: intercomm</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T. 40 41 42 43
Rat fun rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm INTE TYPE INTE	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm EGER, INTENT(IN) :: E(MPI_Comm), INTENT(I)</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C) fd DUT) :: intercomm WT(OUT) :: ierror</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T. 40 41 42 43 44
Rat fun. rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm INTE TYPE INTE	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm EGER, INTENT(IN) :: E(MPI_Comm), INTENT((EGER, OPTIONAL, INTE)</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C) fd DUT) :: intercomm NT(OUT) :: ierror , IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T. 40 41 42 43
Rat fun. rata 10.5.5 MPI_CON IN OUT int MPI_ MPI_Comm INTE TYPE INTE MPI_COMM INTE	<pre>tionale. It would be ni ction explicitly does no ionale.) Another Way to Establi MM_JOIN(fd, intercomm fd intercomm _Comm_join(int fd, MI n_join(fd, intercomm EGER, INTENT(IN) :: E(MPI_Comm), INTENT((EGER, OPTIONAL, INTEN 4_JOIN(FD, INTERCOMM, EGER FD, EGER</pre>	<pre>ce to be able to use MPI_COMM_FREE instead, but that t wait for pending communication to complete. (End of sh MPI Communication) socket file descriptor new intercommunicator (handle) PI_Comm *intercomm) , ierror) BIND(C) fd DUT) :: intercomm NT(OUT) :: ierror , IERROR)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 ³⁹ ticket-248T. 40 41 42 43 44 45

1	MPI_COMM_JOIN is intended for MPI implementations that exist in an environment
2	supporting the Berkeley Socket interface [45, 49]. Implementations that exist in an environ-
3	ment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN
4	and should return MPI_COMM_NULL.

5This call creates an intercommunicator from the union of two MPI processes which are 6 connected by a socket. MPI_COMM_JOIN should normally succeed if the local and remote $\overline{7}$ processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (End of advice to users.)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (End of advice to implementors.)

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable 21byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must 22 not be enabled for the socket. The socket must be in a connected state. The socket must 23be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the 24application to create the socket using standard socket API calls. 25

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not 26return until both processes have called MPI_COMM_JOIN. The two processes are referred 27to as the local and remote processes. 28

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing 29 else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent 30 (see below). 31

If MPI is unable to create an intercommunicator, but is able to leave the socket in its 32 original state, with no pending communication, it succeeds and sets intercomm to 33 MPI_COMM_NULL. 34

The socket must be quiescent before MPI_COMM_JOIN is called and after 35 MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a read on the 36 socket will not read any data that was written to the socket before the remote process called 37 MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a read will not read any data that was 38 written to the socket before the remote process returned from MPI_COMM_JOIN. It is the 39 responsibility of the application to ensure the first condition, and the responsibility of the 40 MPI implementation to ensure the second. In a multithreaded application, the application 41 must ensure that one thread does not access the socket while another is calling 42MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently. 43

44Advice to implementors. MPI is free to use any available communication path(s)45for MPI messages in the new communicator; the socket is only used for the initial 46 handshaking. (End of advice to implementors.) 47

48

Unofficial Draft for Comment Only

8 9

10 11

12

13 14

15

16

17

18

MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of non-	1
MPI communication with pending MPI communication is not defined. Therefore, the result	2
of calling MPI_COMM_JOIN on two connected processes (see Section 10.5.4 on page 416 for	3 4
the definition of connected) is undefined. The returned communicator may be used to establish MPI communication with addi-	4 5
tional processes, through the usual MPI communicator creation mechanisms.	6
tional processes, through the usual WFT communicator creation mechanisms.	7
	8
	9
	10
	11
	12
	13
	14
	15
	16
	17 18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30 31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43 44
	44 45
	45 46
	47
	48

Chapter 11

One-Sided Communications

11.1 Introduction

Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or to update at other processes. [However, processes may not know which data in their own memory need to be accessed or to be updated by remote processes, and may not even know the identity of these processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to periodically poll for potential communication requests to receive and act upon poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form A = B(map), where map is a permutation vector, and A, B and map are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver; and *synchronization* of sender with receiver. The RMA design separates these two functions. [Three communication calls are provided: MPI_PUT (remote write), MPI_GET (remote read) and MPI_ACCUMULATE (remote update). A larger number of synchronization calls are provided that support different synchronization styles. The design is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be imposed by the user, using synchronization calls; the implementation can delay communication operations until the synchronization calls occur, for efficiency.] The following communication calls are provided:

- Remote write: MPI_PUT, MPI_RPUT
- Remote read: MPI_GET, MPI_RGET

0	
7	This chapter refers to an operations set that includes all remote update, remote read and
ticket 270. $^{\rm 8}$	update, and remote atomic swap operations as "accumulate" operations.
9	MPI supports two fundamentally different memory models: separate and unified. The
10	first model makes no assumption about memory consistency and is highly portable. This
11	model is similar to that of weakly coherent memory systems: the user must impose correct
ticket270. 12	ordering of memory accesses through synchronization calls; for efficiency, the implementa-
13	tion can delay communication operations until the synchronization calls occur]. The second
14	model can exploit cache-coherent hardware and hardware-accelerated one-sided operations
ticket270. 15	that are commonly available in high-performance systems. [In this model, communication
16	can be independent of synchronization calls.] The two different models are discussed in
17	detail in Section 11.4. Both models support a large number of synchronization calls to
18	support different synchronization styles.
ticket270. 19	The design of the RMA functions allows implementors to take advantage [, in many
ticket270. 20	cases,] of fast or asynchronous communication mechanisms provided by various platforms,
21	such as coherent or noncoherent shared memory, DMA engines, hardware-supported put/get
ticket270. 22	operations, and communication coprocessors[, etc]. The most frequently used RMA com-
ticket 0.23	munication mechanisms can be layered on top of message-passing. [However, support for
ticket0. 24	asynchronous communication agents in software (handlers, threads, etc.) is needed, for cer-
ticket 270.25	tain RMA functions, in a distributed memory environment.]However, certain RMA functions
26	might need support for asynchronous communication agents in software (handlers, threads,
27	etc.) in a distributed memory environment.
28	We shall denote by origin the process that performs the call, and by target the
29	process in which the memory is accessed. Thus, in a put operation, source=origin and
30	destination=target; in a get operation, source=target and destination=origin.
31	
32	11.2 Initialization
ticket270. 33	
ticket284. 34	[The initialization operation]MPI provides [three] the following window initialization func-
ticket 284. $\frac{35}{22}$	tions, MPI_WIN_CREATE, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED and
36	MPI_WIN_CREATE_DYNAMIC that are collective on an intracommunicator.
ticket270. $\frac{37}{38}$	MPI_WIN_CREATE allows each process [in an intracommunicator group] to specify [, in
ticket270. $\frac{38}{39}$	a collective operation,] a "window" in its memory that is made accessible to accesses by
40	remote processes. The call returns an opaque object that represents the group of processes
41	that own and access the set of windows, and the attributes of each window, as specified
ticket270. $\frac{1}{42}$	by the initialization call. $MPI_WIN_ALLOCATE$ differs from MPI_WIN_CREATE in that
43	the user does not pass allocated memory; $MPI_WIN_ALLOCATE$ returns a pointer to mem-
ticket284. $_{44}^{40}$	ory allocated by the MPI implementation. MPI_WIN_ALLOCATE_SHARED differs from
45	MPI_WIN_ALLOCATE in that the allocated memory can be accessed from all processes in
46	the window's group with direct load/store instructions. Some restrictions apply to the spec-
47	ified communicator. MPI_WIN_CREATE_DYNAMIC creates a window that allows the user
48	to dynamically control which memory is exposed by the window.

• Remote update: MPI_ACCUMULATE, MPI_RACCUMULATE

• Remote read and update: MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP

CHAPTER 11. ONE-SIDED COMMUNICATIONS

• Remote atomic swap operations: MPI_COMPARE_AND_SWAP

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 $\mathbf{5}$

11.2. INITIALIZATION 423				
11.2.1 Wi	ndow Creation		1 2 3	
MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)				
IN	base	initial address of window (choice)	5	
IN	size	size of window in bytes (non-negative integer)	7	
IN	disp_unit	local unit size for displacements, in bytes (positiv	9	
IN	info	info argument (handle)	$^{10}_{11}$ ticket270.	
IN	comm	intra-communicator (handle)	12	
OUT	win	window object returned by the call (handle)	13 14	
int MPI_W	n_create(void *base, MPI MPI_Comm comm, MPI_W	I_Aint size, int disp_unit, MPI_Info in in *win)	15 16 ¹⁷ ticket-248T.	
<pre>MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: base INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size INTEGER, INTENT(IN) :: disp_unit</pre>				
TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Win), INTENT(OUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
<pre>MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)</pre>				
<pre>{static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int disp_unit, const MPI::Info& info, const MPI::Intracomm& comm)(binding deprecated, see Section 15.2) }</pre>				
This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. In C and C++, base is the starting address of a memory region. In Fortran, one can pass ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁸ ³⁸ ³⁹ ticket2				

In C and C++, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous', see also Section 16.2.12 on page 673). A process may elect to expose no memory by specifying size = 0.

The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor disp_unit specified by the target process, at window creation.

Rationale. The window size is specified using an address sized integer, so as to allow windows that span more than 4 GB of address space. (Even if the physical memory

Unofficial Draft for Comment Only

40

41

42

43

44

4546

47

	424 CHAPTER 11. ONE-SIDED COMMUNICATIONS
1 2	size is less than 4 GB, the address range may be larger than 4 GB, if addresses are not contiguous.) (<i>End of rationale.</i>)
3 4 5 6 7 8	Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax) sizeof(type), for a window that consists of an array of elements of type type. The later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (<i>End of advice to users.</i>)
9 10 ticket270. 11	The info argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info key[is]s are predefined:
12 13 ticket270. $_{14}$ 15 ticket270. $_{16}$	no_locks — if set to true, then the implementation may assume that the local window is never locked (by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL). This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.
17 18 19	$accumulate_ordering$ — controls the ordering of accumulate operations at the target. See Section 11.7.2 for details.
20 21 22 23 24 25	accumulate_ops — if set to same_op, the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation. If set to same_op_no_op, then the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation or MPI_NO_OP. This can eliminate the need to protect access for certain operation types where the hardware can guarantee atomicity. The default is same_op_no_op.
ticket270. ²⁶ 27	
28 29 30 31	Advice to users. If windows are passed to libraries, the user needs to ensure that the info keys specified at window creation are communicated to the called library, which might need to constrain the operations on the passed window. (End of advice to users.)
32 33 34 35 36 37 ticket270. 38	The various processes in the group of comm may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the get, put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to [erroneous]undefined results.
ticket270. 39 40 41 42 43 44 45 46 47 48	<i>Rationale.</i> The reason for specifying the memory that may be accessed from another process in an RMA operation is to permit the programmer to specify what memory can be a target of RMA operations and for the implementation to enforce that specification. For example, with this definition, a server process can safely allow a client process to use RMA operations, knowing that (under the assumption that the MPI implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (<i>End of rationale.</i>)

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM (Section 8.2, page 353) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (End of advice to users.)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

Implementors should document any performance impact of window alignment. (*End of advice to implementors.*)

11.2.2 Window That Allocates Memory

MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)

IN	size	size of window in bytes (non-negative integer)	29
		· · · · · · · · · · · · · · · · · · ·	30
IN	disp_unit	local unit size for displacements, in bytes (positive in-	31
		teger)	32
IN	info	info argument (handle)	33
IN	comm	intra-communicator (handle)	34
	hacentu		35
OUT	baseptr	initial address of window (choice)	36
OUT	win	window object returned by the call (handle)	37
			38
int MPI_	Win_allocate(MPI_Aint siz	e, int disp_unit, MPI_Info info,	39
	MPI_Comm comm, void	<pre>*baseptr, MPI_Win *win)</pre>	40
		•	$_{41}$ ticket-248T.
	•	<pre>info, comm, baseptr, win, ierror) BIND(C)</pre>	42
	INTRINSIC :: ISO_C_BINE		43
	GER(KIND=MPI_ADDRESS_KIND		44
INTEGER, INTENT(IN) :: disp_unit 45			45
	(MPI_Info), INTENT(IN) ::		46
	(MPI_Comm), INTENT(IN) ::		47
TYPE(C_PTR), INTENT(OUT) :: baseptr			48

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12

13

14

15 16

17

18

19

20

21

22

 24

25 26 27

28

₂₃ ticket270.

1	TYPE(MPI_Win), INTENT(OUT) :: win
2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
4 5	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
6	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
7	This is a collective call executed by all processes in the group of comm. On each
8	This is a collective call executed by all processes in the group of comm . On each process, it allocates memory of at least size size bytes, returns a pointer to it, and returns a
9	window object that can be used by all processes in comm to perform RMA operations. The
10	returned memory consists of size bytes local to each process, starting at address baseptr
11	and is associated with the window as if the user called MPI_WIN_CREATE on existing
12	memory. The size argument may be different at each process and $size = 0$ is valid; however, a
13	library might allocate and expose more memory in order to create a fast, globally symmetric
14	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
15	Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2
ticket229.5. 16	for an explanation of the type used for baseptr .
17	If the Fortran compiler provides TYPE(C_PTR), then the following interface must be
18	provided in the \mathtt{mpi} module and should be provided in $\mathtt{mpif.h}$ through overloading, i.e., with
19	the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR,
20 21	but with a different linker name:
21	INTERFACE MPI_WIN_ALLOCATE
23	SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
24	WIN, IERROR)
25	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
26	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
27	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
28	TYPE(C_PTR) :: BASEPTR
29	END SUBROUTINE
30	END INTERFACE
31	
32	The linker name base of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The
33	implied linker names are described in Section $16.2.5$ on page 651 .
34 35	Rationale. By allocating (potentially aligned) memory instead of allowing the user
36	to pass in an arbitrary buffer, this call can improve the performance for systems with
ticket270. 37	remote direct memory access significantly. This also permits the collective allocation
38	of memory and supports what is sometimes called the "symmetric allocation" model
39	that can be more scalable (for example, the implementation can arrange to return
40	an address for the allocated memory that is the same on all processes). $(End \ of$
41	rationale.)
42	
43	The info argument can be used to specify hints similar to the info argument for
44	MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:
45	same_size — if set to true, then the implementation may assume that the argument size is
46	identical on all processes.
ticket284. $^{47}_{48}$	
48	

11.2.3 Window That Allocates Shared Memory		1	
			2
			3
MPI_WI	ALLOCATE_SHAR	ED(size, info, comm, baseptr, win)	4
IN	size	size of local window in bytes (non-negative integer)	5
			6
IN	info	info argument (handle)	7
IN	comm	intra-communicator (handle)	8 9
OUT	baseptr	address of local allocated window segment (choice)	10
OUT	win	window object returned by the call (handle)	11
			12
int MPT	Win allocate shar	red(MPI_Aint size, MPI_Info info, MPI_Comm comm,	13
		tr, MPI_Win *win)	14
NDT	-		15 ticket-248T.
		size, info, comm, baseptr, win, ierror) BIND(C)	16
		SO_C_BINDING, ONLY : C_PTR ress_kind), INTENT(IN) :: size	17
	EGER(KIND-MFI_Add E(MPI_Info), INTE		18
	E(MPI_Comm), INTER		19
TYPE(C_PTR), INTENT(OUT) :: baseptr			20 21
	E(MPI_Win), INTEN		22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			23
MDT LITN	ALLOCATE CUADED (SIZE, INFO, COMM, BASEPTR, WIN, IERROR)	24
			25
INTEGER INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR			26
			27
		executed by all processes in the group of comm. On each	28
-	· · ·	y of at least size size bytes that is shared among all processes	29
		er to the locally allocated segment in baseptr that can be used	30
		e calling process. The locally allocated memory can be the by remote processes; the base pointers for other processes can	31
-		MPI_WIN_SHARED_QUERY. The call also returns a window	32
-	e e e e e e e e e e e e e e e e e e e	all processes in comm to perform RMA operations. The size	33 34
-		z each process and size = 0 is valid; however, a library might	35
		mory in order to create a fast, globally symmetric allocation.	36
	—	to ensure that the communicator comm represents a group of	37
processes	s that can create a sl	hared memory segment that can be accessed by all processes	38
_		of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in	39
		I_WIN_ALLOCATE_SHARED; in particular, see the rationale	40
in Section 8.2 for an explanation of the type used for baseptr. The allocated memory			41

is contiguous across process ranks unless the info key alloc_shared_noncontig is specified. Contiguous across process ranks means that the first address in the memory segment of process i is consecutive with the last address in the memory segment of process i-1. This may enable the user to calculate remote address offsets with local information only.

If the Fortran compiler provides TYPE(C_PTR), then the following interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with

 $_{45}$ ticket 229.5.

48

42

43

44

46

1 2		outine name as the routine w different linker name:	vith INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR,		
3 4 5 6 7 8 9	INTERFACE MPI_WIN_ALLOCATE_SHARED SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR INTEGER :: INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE				
10		YPE(C_PTR) :: BASEPTR			
11	END SUBROUTINE END INTERFACE				
12					
13 14	The linker name base of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied linker names are described in Section 16.2.5 on page 651.				
15 16 17 18 19	The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE, MPI_WIN_ALLOC, and MPI_ALLOC_MEM. The additional info key alloc_shared_noncontig allows the library to optimize the layout of the shared memory segments in memory.				
20 21 22 23 24	Advice to users. If the info key alloc_shared_noncontig is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (End of advice to users.)				
25 26 27 28 29 30	Advice to implementors. If the user sets the info key alloc_shared_noncontig to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (End of advice to implementors.)				
31 32 33 34 35 36 37	user progra memory m Section 11 MPI_WIN_	am depends on the architectur nodel (see Section 11.4) by ut .5) or explicitly completing o	ses from/to the shared memory as observed by the re. A consistent view can be created in the unified filizing the window synchronization functions (see utstanding store accesses (e.g., by calling e semantics for accessing shared memory windows		
38 39	MPI_WIN	_SHARED_QUERY(win, rank, s	size, baseptr)		
39 40	IN	win	shared memory window object (handle)		
41	IN	rank	rank in the group of window win (non-negative inte-		
42 43	11 N	TUTIN	ger)		
43	OUT	size	size of the window segment (non-negative integer)		
$45 \\ 46$	OUT	baseptr	address for load/store access to window segment (choice) $% \left({{{\rm{choice}}} \right)$		
40	int MPT W	in shared query(MPT Win s	win, int rank, MPI_Aint *size,		
48		void *baseptr)	, I a, in, in		

ticket-248T.		1
	MPI_Win_shared_query(win, rank, size, baseptr, ierror) BIND(C)	2
	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	3
	TYPE(MPI_Win), INTENT(IN) :: win	4
	INTEGER, INTENT(IN) :: rank	5
	<pre>INTEGER(KIND=MPI_Address_kind), INTENT(IN) :: size</pre>	6
	TYPE(C_PTR), INTENT(OUT) :: baseptr	7
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
	MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, BASEPTR, IERROR)	9
	INTEGER WIN, RANK, IERROR	10
	INTEGER WIN, RANK, TERROR INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	11
	INTEGER (KIND-MFI_ADDRESS_KIND) SIZE, BASEFIN	12
	This function queries the process-local address for remote memory segments created	13
	with $MPI_WIN_ALLOCATE_SHARED$. This function can return different process-local ad-	14
	dresses for the same physical memory on different processes. The returned memory can be	15
	used for load/store accesses subject to the constraints defined in Section 11.7. This function	16
	can only be called with windows of type $MPI_WIN_FLAVOR_SHARED.$ If the passed window	17
	is not of flavor $MPI_WIN_FLAVOR_SHARED$, the error $MPI_ERR_RMA_WRONG_FLAVOR$ is	18
	raised. When rank is MPI_PROC_NULL, the pointer and size returned are the pointer and	19
	size of the memory segment belonging the lowest rank that specified $size > 0$. If all processes	20
	in the group attached to the window specified $size = 0$, then the call returns $size = 0$ and	21
	a baseptr as if MPI_ALLOC_MEM was called with size $= 0$.	22 ticket 229.5.
	If the Fortran compiler provides TYPE(C_PTR), then the following interface must be	23
	provided in the mpi module and should be provided in mpif.h through overloading, i.e., with	24
l	the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR,	25
l	but with a different linker name:	26
		27
l	INTERFACE MPI_WIN_SHARED_QUERY	28
l	SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, BASEPTR, IERROR)	29
l	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	30
l	INTEGER :: WIN, RANK, IERROR	31
	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	32
l	TYPE(C_PTR) :: BASEPTR	33
	END SUBROUTINE	34
l	END INTERFACE	35
4		

The linker name base of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied linker names are described in Section 16.2.5 on page 651.

11.2.4 Window of Dynamically Attached Memory

The MPI-2 RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make one-sided access to 44such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using malloc or new respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and

Unofficial Draft for Comment Only

36

37

38 39

40 41

42

43

4546

47

48

ticket270.

1then implement routines for allocating memory from within that memory. In addition, there $\mathbf{2}$ is no easy way to handle the situation where the predefined amount of memory turns out to 3 be inadequate. To support this model, the routine MPI_WIN_CREATE_DYNAMIC creates a 4 window that makes it possible to expose memory without remote synchronization. It must $\mathbf{5}$ be used in combination with the local routines MPI_WIN_ATTACH and MPI_WIN_DETACH. 6 7 MPI_WIN_CREATE_DYNAMIC(info, comm, win) 8 9 IN info argument (handle) info 10 IN comm intra-communicator (handle) 11 OUT win window object returned by the call (handle) 1213 int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win) 14ticket-248T. 15 MPI_Win_create_dynamic(info, comm, win, ierror) BIND(C) 16TYPE(MPI_Info), INTENT(IN) :: info 17TYPE(MPI_Comm), INTENT(IN) :: comm 18 TYPE(MPI_Win), INTENT(OUT) :: win 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR) 21INTEGER INFO, COMM, WIN, IERROR 22 23This is a collective call executed by all processes in the group of comm. It returns 24a window win without memory attached. Existing process memory can be attached as 25described below. This routine returns a window object that can be used by these processes to 26perform RMA operations on attached memory. Because this window has special properties, 27it will sometimes be referred to as a *dynamic* window. 28The info argument can be used to specify hints similar to the info argument for 29 MPI_WIN_CREATE. 30 In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for 31 all RMA functions is the address at the target; i.e., the effective window_base is MPI_BOTTOM 32 and the disp_unit is one. Users should use MPI_GET_ADDRESS at the target process to 33 determine the address of a target memory location and communicate this address to the 34 origin process. 35 36 Advice to implementors. In environments with heterogeneous data representations, 37 care must be exercised in communicating addresses between processes. For example, 38 it is possible that an address valid at the target process (for example, a 64-bit pointer) 39 cannot be expressed as an address at the origin (for example, the origin uses 32-bit 40 pointers). For this reason, a portable MPI implementation should ensure that the 41 type MPI_AINT (cf. Table 3.3 on Page 31) is able to store addresses from any process. 42(End of advice to implementors.) 43 Memory in this window may not be used as the target of one-sided accesses in this 4445window until it is attached using the function MPI_WIN_ATTACH. That is, in addition to

⁴⁶ using MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use

⁴⁷ MPI_WIN_ATTACH before any local memory may be the target of an MPI RMA operation.
 ⁴⁸ Only memory that is currently accessible may be attached.

MPI_WIN_ATTACH(win, base, size)

IN	win	window object (handle)
IN	base	initial address of memory to be attached
IN	size	size of memory to be attached in bytes

int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)

<pre>MPI_Win_attach(win, base, size, ierror) BIND(C)</pre>	
TYPE(MPI_Win), INTENT(IN) :: win	
TYPE(*), ASYNCHRONOUS :: base	
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size</pre>	Э
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR) INTEGER WIN, IERROR	

<type> [base]BASE(*) INTEGER (KIND=MPI_ADDRESS_[SIZE]KIND) [size]SIZE

Attaches a local memory region beginning at **base** for remote access within the given window. The memory region specified must not contain any part that is already attached to the window win, that is, attaching overlapping memory concurrently within the same window is erroneous. The argument win must be a window that was created with MPI_WIN_CREATE_DYNAMIC. Multiple (but non-overlapping) memory regions may be attached to the same window.

Requiring that memory be explicitly attached before it is exposed to Rationale. one-sided access by other processes can significantly simplify implementations and improve performance. The ability to make memory available for RMA operations without requiring a collective MPI_WIN_CREATE call is needed for some one-sided programming models. (End of rationale.)

Advice to users. [Memory registration] Attaching memory to a window may require the use of scarce resources; thus, attaching large regions of memory is not recommended in portable programs. [Memory registration] Attaching memory to a window may fail if sufficient resources are not available; this is similar to the behavior of MPI_ALLOC_MEM.

The user is also responsible for ensuring that [memory registration] MPI_WIN_ATTACH at the target has **completed** returned before a process attempts to target that memory with an MPI RMA call.

Performing an RMA operation to memory that has not been attached from to a window created with MPI_WIN_CREATE_DYNAMIC is erroneous. (End of advice to users.)

Advice to implementors. A high-quality implementation will attempt to make as much memory available for registration attaching as possible. Any limitations should be documented by the implementor. (End of advice to implementors.)

Unofficial Draft for Comment Only

1

9

16

20

21

22

23

2425

26

27

28

29

30 31

33

35

36

39 40

41

42

43 44

46

 $_{32}$ ticketxx11/1

 $_{34}$ ticketxx11/1

 37 ticketxx/11/

 38 ticketxx/11/

ticketxx:5/11

 45 ticketxx/11/

⁴⁷ ticket0.

1 [Memory registration] Attaching memory is a local operation as defined by MPI, which $\mathbf{2}$ means that the call is not collective and completes without requiring any MPI routine to be 3 called in any other process. Memory may be detached with the routine MPI_WIN_DETACH. 4 After memory has been detached, it may not be the target of an MPI RMA operation on $\mathbf{5}$ that window (unless the memory is re-attached with MPI_WIN_ATTACH). 6 7 MPI_WIN_DETACH(win, base) 8 9 IN window object (handle) win 10 IN initial address of memory to be detached base 11 12ticket 140a. $_{13}$ int MPI_Win_detach(MPI_Win win, const void *base) ticket-248T. $_{14}$ MPI_Win_detach(win, base, ierror) BIND(C) 15TYPE(MPI_Win), INTENT(IN) :: win 16TYPE(*), ASYNCHRONOUS :: base 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI_WIN_DETACH(WIN, BASE, IERROR) 19 INTEGER WIN, IERROR 20txx:12/9/11.₂₁ <type> [base]BASE(*) 22 Detaches a previously attached memory region beginning at base. The arguments base 23and win must match the arguments passed to a previous call to MPI_WIN_ATTACH. 24 25Advice to users. Detaching memory may permit the implementation to make more 26efficient use of special memory or provide memory that may be needed by a subsequent 27MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. 28Memory should be detached before it is freed by the user. (End of advice to users.) 29 30 Memory becomes detached when the associated dynamic memory window is freed, see 31 Section 11.2.5. 32 ticket270. ³³ 11.2.5 Window Destruction 34 35 36 MPI_WIN_FREE(win) 37 INOUT window object (handle) win 38 39 int MPI_Win_free(MPI_Win *win) 40 ticket-248T. 41 MPI_Win_free(win, ierror) BIND(C) 42TYPE(MPI_Win), INTENT(INOUT) :: win 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4445MPI_WIN_FREE(WIN, IERROR) 46INTEGER WIN, IERROR 47{void MPI::Win::Free() (binding deprecated, see Section 15.2) } 48

Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: [i.e.]e.g., the process has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. [When the call returns, the window memory can be freed.]The memory associated with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_CREATE_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by a call to MPI_WIN_DETACH.

Advice to implementors. MPI_WIN_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win called free. This[,] is to ensure that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. The only exception to this rule is

when the user sets the no_locks info [argument]key to true when creating the window. In that case, the local window can be freed without barrier synchronization. (*End of advice to implementors.*)

11.2.6 Window Attributes

The following [three] attributes are cached with a window[,] when the window is created.

		20
MPI_WIN_BASE MPI_WIN_SIZE	window base address.	$^{27}_{_{28}}$ ticket270.
MPI_WIN_DISP_UNIT	displacement unit associated with the window.	29 ticket 270.
[ticket270.]MPI_WIN_CREATE_FLAVOR	how the window was created.	30 ticket 270.
[ticket270.]MPI_WIN_MODEL	memory model for window.	31
	MDL WIN BASE Share Silar)	32
In C, calls to MPI_Win_get_attr(win,	,	33
MPI_Win_get_attr(win, MPI_WIN_SIZE, &		$_{34}$ ticket270.
MPI_Win_get_attr(win, MPI_WIN_DISP_U		$_{35}$ ticket0.
MPI_Win_get_attr(win, MPI_WIN_CREAT	E_FLAVOR, &create_kind, &flag), and	$_{36}$ ticket270.
MPI_Win_get_attr(win, MPI_WIN_MODE	L, &memory_model, &flag) will return in base ${ m a}$	37
pointer to the start of the window win, an	nd will return in size[and], disp_unit, create_kind,	$_{38}$ ticket 270.
	nd], displacement unit of the window, the kind of	$_{39}$ ticket270.
routine used to create the window, and the	e memory model, respectively. [And similarly, in	$_{40}$ ticket270.
C++.]And similarly, in C++ (binding de	precated, see Section 15.2). A detailed listing of	$_{41}$ ticket270.
the type of the pointer in the attribute v	value argument to MPI_Win_get_attr and	$_{42}$ ticket270.
MPI_Win_set_attr is shown in Table 11.1.		$_{43}$ ticket 283.
In Fortran, calls to $MPI_WIN_GET_A$	ATTR(win, MPI_WIN_BASE, base, flag, ierror),	44
MPI_WIN_GET_ATTR(win, MPI_WIN_SIZ	ZE, size, flag, ierror)[and],	$_{45}$ ticket 270.
MPI_WIN_GET_ATTR(win, MPI_WIN_DIS	SP_UNIT, disp_unit, flag, ierror),	$_{46}$ ticket 270.
MPI_WIN_GET_ATTR(win, MPI_WIN_CR	EATE_FLAVOR, create_kind, flag, ierror), and	47

MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in

1

 $\mathbf{2}$

3

5

6

8 9

10

12 13

14

15

16

20

21 22

23 24

48

 4 ticket0.

 7 ticket270.

 11 ticket270.

¹⁷ ticket270.

¹⁸ ticket270.

₂₅ ticket270.

 $_{26}$ ticket270.

¹⁹ ticketxx:5/11

```
434
                                                      CHAPTER 11. ONE-SIDED COMMUNICATIONS
            1
                                         Attribute
                                                                     C Type
            \mathbf{2}
                                         MPI_WIN_BASE
                                                                     void *
            3
                                         MPI_WIN_SIZE
                                                                     MPI_Aint *
            4
                                         MPI_WIN_DISP_UNIT
                                                                     int *
            5
                                         MPI_WIN_CREATE_FLAVOR
                                                                     int *
            6
                                         MPI_WIN_MODEL
                                                                     int *
            7
            8
                 Table 11.1: C types of attribute value argument to MPI_Win_get_attr and
            9
                 MPI_Win_set_attr.
            10
            11
           12
                 base, size and, disp_unit create_kind and memory_model the (integer representation of) the
                                                                                                              ticket270.
  ticket 270. ^{13}
                                                                                                               ticket270.
                 base address, the size and, the displacement unit of the window win, the kind of routine
  ticket270.<sup>14</sup>
                 used to create the window, and the memory model, respectively.
  ticket270.<sup>15</sup>
                     The values of create_kind are
           16
           17
                 MPI_WIN_FLAVOR_CREATE
                                                         Window was created with MPI_WIN_CREATE.
           18
                 MPI_WIN_FLAVOR_ALLOCATE
                                                         Window was created with MPI_WIN_ALLOCATE.
           19
                 MPI_WIN_FLAVOR_DYNAMIC
                                                         Window was created with
                                                         MPI_WIN_CREATE_DYNAMIC.
  ticket284. 20
           21
                 MPI_WIN_FLAVOR_SHARED
                                                         Window was created with
                                                         MPI_WIN_ALLOCATE_SHARED.
           22
           23
                      The values of memory_model are MPI_WIN_SEPARATE and MPI_WIN_UNIFIED. The mean-
           ^{24}
                 ing of these is described in Section 11.4.
           25
                      In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address
           26
                 is MPI_BOTTOM and the size is 0. In C, pointers are returned and in Fortran, the values are
           27
                 returned, for the respective attributes. (The window attribute access functions are defined
            28
  ticket270.
                 in Section 6.7.3, page 284.) The value returned for an attribute on a window is constant
            29
                 over the lifetime of the window.
           30
                      The other "window attribute," namely the group of processes attached to the window,
           ^{31}
                 can be retrieved using the call below.
           32
           33
           34
                 MPI_WIN_GET_GROUP(win, group)
           35
                   IN
                                                         window object (handle)
                             win
           36
                   OUT
                             group
                                                         group of processes which share access to the window
           37
                                                         (handle)
           38
           39
            40
                 int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
ticket-248T. 41
                 MPI_Win_get_group(win, group, ierror) BIND(C)
           42
                      TYPE(MPI_Win), INTENT(IN) :: win
           43
                      TYPE(MPI_Group), INTENT(OUT) :: group
           44
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           45
           46
                 MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
           47
                      INTEGER WIN, GROUP, IERROR
            48
```

1

{MPI::Group MPI::Win::Get_group() const(binding deprecated, see Section 15.2) }

MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window[.] associated with win. The group is returned in group.

11.3 Communication Calls

MPI supports [three]the following RMA communication calls: MPI_PUT [transfers]and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET [transfers]and MPI_RGET transfer data from the target memory to the caller memory; [and] MPI_ACCUMULATE [updates]and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory[.]; MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE and MPI_FETCH_AND_OP atomically return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and at the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.5, page 454. Transfers can also be completed with calls to flush routines; see Section 11.5.4, page 467 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3, page 58.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call[,] until the [subsequent synchronization call completes.]operation completes at the origin.

[It is erroneous to have concurrent conflicting accesses to the same memory location in a window]The outcome of concurrent conflicting accesses to the same memory locations is undefined; if a location is updated by a put or accumulate operation, then [this location cannot be accessed by a load or another RMA operation]the outcome of [local] loads or other RMA operations is undefined until the updating operation has completed at the target. There is one exception to this rule; namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred in some order. In addition, [if a window cannot concurrently be updated by a put or accumulate operation and by a local store operation. This, even if these two updates access different locations in the window. The last restriction enables more efficient implementations of RMA operations on many systems.]the outcome of concurrent [local]load/store and RMA updates to the same memory location is undefined. These restrictions are described in more detail in Section 11.7, page 471.

The calls use general datatype arguments to specify communication buffers at the origin and at the target. Thus, a transfer operation may also gather data at the source and scatter it at the destination. However, all arguments specifying both communication buffers are provided by the caller.

For all [three] RMA calls, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory.

Rationale. The choice of supporting "self-communication" is the same as for messagepassing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (*End of rationale.*)

Unofficial Draft for Comment Only

2 3 ticket0. 56 7 * ticket270. 9 ticket270. ¹⁰ ticket270. ¹¹ ticket270. $_{12}$ ticket 270. $_{13}$ ticket 270. $_{14}$ ticket 270. 151617 18 ¹⁹ ticket 270. 20212223²⁴ ticket270. 25 ticket270. ²⁶ ticket270. 27²⁸ ticket270. ²⁹ ticket284. 30 3132 ³³ ticket270. 34 35 ³⁶ ticket284. 37 38 39 40 41 42⁴³ ticket270. 44 4546 4748

ticket 270. $^{\rm 1}$	MPI_PROC_NULL is a valid target rank in [the MPI RMA calls MPI_ACCUMULATE,				
2	MPI_GET, and MPI_PUT]all MPI RMA communication calls. The effect is the same a				
3	for MPI_PROC_NULL in MPI point-to-point communication. After any RMA operation with				
4 5	rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization				
6	method that started the epoch.				
7	11.3.1 Put				
	The execution of a put operation is similar to the execution of a send by the originand a matching receive by the target process. The obvious difference is that all are provided by one call — the call executed by the origin process.				
11					
12					
13 14	³ MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target				
15		target_datatyp	,		
16	IN	origin_addr	initial address of origin buffer (choice)		
17 18	IN	origin_count	number of entries in origin buffer (non-negative integer)		
19	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
20	IN	target_rank	rank of target (non-negative integer)		
21	IN	2			
22 23	IN	target_disp	displacement from start of window to target buffer (non-negative integer)		
24 25	IN	target_count	number of entries in target buffer (non-negative integer)		
26		tannat datat sa			
27	IN	target_datatype	datatype of each entry in target buffer (handle)		
28	IN	win	window object used for communication (handle)		
29		_ /			
ticket140. 30 31	3 1 3 1				
31	origin_addabypo, into bargoo_ram, in r_nino bargoo_arop, into				
ticket229.2. 33	<pre>target_count, MPI_Datatype target_datatype, MPI_Win win)</pre>				
ticket-248T. $_{34}$	MPI_Put		<pre>in_count, origin_datatype, target_rank,</pre>		
35			<pre>target_count, target_datatype, win, ierror)</pre>		
36	TUD	BIND(C)			
37	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count				
38					
39	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp				
40		E(MPI_Win), INTENT			
41 42			TENT(OUT) :: ierror		
42					
44	MP1_P01		IN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET COUNT TARGET DATATYPE WIN IERROR)		
45	TRIGET_DIDI, TRIGET_DOUNT, TRIGET_DATATILE, WIN, TEMODI				
46	U	▲	ESS_KIND) TARGET_DISP		
47			ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,		
48		.GET_DATATYPE, WIN,			

Transfers origin_count successive entries of the type specified by the origin_datatype, starting at address origin_addr on the origin node to the target node specified by the win, target_rank pair. The data are written in the target buffer at address target_addr = window_base + target_disp×disp_unit, where window_base and disp_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target_count and target_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.

The communication must satisfy the same constraints as for a similar message-passing communication. The target_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window or in attached memory in a dynamic window.

The target_datatype argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the target datatype object was defined at the target process[,] by the same sequence of calls used to define it at the origin process. The target datatype must contain only relative displacements, not absolute addresses. The same holds for get and accumulate. [In the case of windows created with MPI_WIN_CREATE_DYNAMIC, displacements in the target datatype must be relative to MPI_BOTTOM.]

Advice to users. The target_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment[,] if only portable datatypes are used (portable datatypes are defined in Section 2.4, page 11).

The performance of a put transfer can be significantly affected, on some systems, [from]by the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI_ALLOC_MEM or MPI_WIN_ALLOCATE may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This, both for debugging purposes, and for protection with client-server codes that use RMA. I.e., a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurred. Note that the condition can be checked at the origin. Of course,

Unofficial Draft for Comment Only

1

2

3

4

5 6

7

8

9

10

11

12

13

14

15

16

18

19

20

21

23

 24

28

29 30

31

32

35

36

38

40

41 42

43

44

45

46

47

48

₁₇ ticket270.

 $_{22}$ ticket 270.

 $_{25}$ ticket 270.

₂₇ ticket308.

 $_{_{34}}$ ticket270.

37 ticket270.

39 ticket270.

ticket270.

		400		CHAI TER II. ONE-SIDED COMMONICATIONS
	1 2 3		added safety achieved by h checks. (<i>End of advice</i>	y such checks has to be weighed against the added cost of to implementors.)
	4 5 6	11.3.2	Get	
	7 8	MPI_GET	Г(origin_addr, origin_coun target_datatype, v	nt, origin_datatype, target_rank, target_disp, target_count, vin)
	9 10	OUT	origin_addr	initial address of origin buffer (choice)
	11 12	IN	origin_count	number of entries in origin buffer (non-negative inte- ger)
	13 14	IN	origin_datatype	datatype of each entry in origin buffer (handle)
	15	IN	target_rank	rank of target (non-negative integer)
	16 17	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)
	18 19 20	IN	target_count	number of entries in target buffer (non-negative integer)
	21	IN	target_datatype	datatype of each entry in target buffer (handle)
	22 23	IN	win	window object used for communication (handle)
ticket229.2 ticket-248T		MPI_Get(TYPE INTE TYPE INTE TYPE INTE	<pre>MPI_Aint target MPI_Datatype ta (origin_addr, origin_ target_disp, ta BIND(C) E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE EGER(KIND=MPI_ADDRESS E(MPI_Win), INTENT(IN) EGER, OPTIONAL, INTEN</pre>	<pre>crigin_datatype, int target_rank, t_disp, int target_count, arget_datatype, MPI_Win win) count, origin_datatype, target_rank, arget_count, target_datatype, win, ierror) ASYNCHRONOUS :: origin_addr origin_count, target_rank, target_count ENT(IN) :: origin_datatype, target_datatype S_KIND), INTENT(IN) :: target_disp () :: win T(OUT) :: ierror</pre>
	 39 40 41 42 43 44 45 46 	<typ INTF INTF TARC</typ 	TARGET_DISP, TA be> ORIGIN_ADDR(*) EGER(KIND=MPI_ADDRESS EGER ORIGIN_COUNT, OR GET_DATATYPE, WIN, IE PI::Win::Get(void *or	IGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
	47 48		MPI::Aint targe	et_disp, int target_count,

1 const MPI::Datatype& target_datatype) const(binding deprecated, $\mathbf{2}$ see Section 15.2 } 3 Similar to MPI_PUT, except that the direction of data transfer is reversed. Data 4 are copied from the target memory to the origin. The origin_datatype may not specify 5 overlapping entries in the origin buffer. The target buffer must be contained within the 6 $_{7}$ ticket270. target window or within attached memory in a dynamic window, and the copied data must fit, without truncation, in the origin buffer. 8 9 11.3.3 Examples for Communication Calls 10 ticket270. 11 These examples show the use of the MPI_GET function. As all MPI RMA communication 12functions are nonblocking, they must be completed. In the following, this is accomplished 13 with the routine MPI_WIN_FENCE, introduced in Section 11.5. 1415**Example 11.1** We show how to implement the generic indirect assignment A = B(map), 16where A, B and map have the same distribution, and map is a permutation. To simplify, we 17assume a block distribution with equal size blocks. 18 SUBROUTINE MAPVALS(A, B, map, m, comm, p) 19 202122 23INTEGER otype(p), oindex(m), & ! used to construct origin datatypes 24& ! used to construct target datatypes 2526& 27INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal 28 29 ! This part does the work that depends on the locations of B. 30 ! Can be reused while this does not change 3132 CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr) 33 CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL, 34 & comm, win, ierr) 35 36 37

```
! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change
! Compute number of entries to be received from each process
```

DO i=1,p count(i) = 0END DO DO i=1.m j = map(i)/m+1count(j) = count(j)+1

USE MPI

REAL A(m), B(m)

win, ierr

INTEGER m, map(m), comm, p

ttype(p), tindex(m),

count(p), total(p),

38

39 40 41

42

43

44

45

46

47

48

```
1
     END DO
\mathbf{2}
3
     total(1) = 0
4
     DO i=2,p
\mathbf{5}
      total(i) = total(i-1) + count(i-1)
6
     END DO
7
8
     DO i=1,p
9
       count(i) = 0
10
     END DO
11
12
     ! compute origin and target indices of entries.
13
     ! entry i at current process is received from location
14
     ! k at process (j-1), where map(i) = (j-1)*m + (k-1),
15
     ! j = 1...p and k = 1...m
16
17
     DO i=1,m
18
       j = map(i)/m+1
19
       k = MOD(map(i), m) + 1
20
       count(j) = count(j)+1
21
       oindex(total(j) + count(j)) = i
22
       tindex(total(j) + count(j)) = k
23
     END DO
^{24}
25
     ! create origin and target datatypes for each get operation
26
     DO i=1,p
27
       CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, oindex(total(i)+1),
                                                                                   &
28
                                             MPI_REAL, otype(i), ierr)
29
       CALL MPI_TYPE_COMMIT(otype(i), ierr)
30
       CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, tindex(total(i)+1),
                                                                                   &
^{31}
                                             MPI_REAL, ttype(i), ierr)
32
       CALL MPI_TYPE_COMMIT(ttype(i), ierr)
33
     END DO
34
     ! this part does the assignment itself
35
36
     CALL MPI_WIN_FENCE(0, win, ierr)
37
     DO i=1,p
38
       CALL MPI_GET(A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
39
     END DO
40
     CALL MPI_WIN_FENCE(0, win, ierr)
41
42
     CALL MPI_WIN_FREE(win, ierr)
43
     DO i=1,p
44
       CALL MPI_TYPE_FREE(otype(i), ierr)
45
       CALL MPI_TYPE_FREE(ttype(i), ierr)
46
     END DO
47
     RETURN
48
     END
```

Example 11.2

A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,
                                                                  &
                    comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i), m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. This will allow, for example, the accumulation of a sum by having all involved processes add their contribution to the sum variable in the memory of one process. The accumulate functions have slightly different semantics than the put and get functions; see Section 11.7 for details.

```
Accumulate Function
```

```
25
26
27
28
29
30
31
32
<sub>33</sub> ticket270.
_{34} ticket 270.
35
36
37
38
39
40
41
42
```

 24

	442		CHAPTER 11	. ONE-SIDED COMMUNICATIONS			
1 2	MPI_ACC	MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)					
3	IN	origin_addr	initial addr	ess of buffer (choice)			
4 5	IN	origin_count	number of	entries in buffer (non-negative integer)			
6	IN	origin_datatype	datatype o	f each entry (handle)			
7	IN	target_rank	rank of tar	get (non-negative integer)			
8 9 10	IN	target_disp	-	nt from start of window to beginning of tar- non-negative integer)			
11 12	IN	target_count	-	entries in target buffer (non-negative inte-			
13	IN	target_datatype	- ,	f each entry in target buffer (handle)			
14 15	IN	op		ration (handle)			
16	IN	win		ject (handle)			
17							
ticket 140. 18	int MPI	<pre>int MPI_Accumulate(const void *origin_addr, int origin_count,</pre>					
19 20	MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)						
21							
ticket229.2. 22		MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,					
ticket-248T. 23	MPI_Accı						
24 25		<pre>target_disp, target_count, target_datatype, op, win, ierror) BIND(C)</pre>					
26	TYPI	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>					
27				arget_rank, target_count			
28				in_datatype, target_datatype			
29 30		EGER(KIND=MPI_ADDRES E(MPI_Op), INTENT(IN		IN) :: target_disp			
30	TYPE(MPI_Win), INTENT(IN) :: win						
32		INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
33	MPI_ACCU	MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,					
34	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR						
35 36	01	pe> ORIGIN_ADDR(*)					
37		EGER(KIND=MPI_ADDRES					
38		GER ORIGIN_COUNT, U GET_DATATYPE, OP, WI		ARGET_RANK, TARGET_COUNT,			
39							
40 41	{void M		-	<pre>in_addr, int origin_count, const , int target_rank, MPI::Aint</pre>			
41 42				const MPI::Datatype&			
43			-	& op) const(binding deprecated, see			
44		Section 15.2) }	-				
45	Accu	umulate the contents of	the origin buffer (as	defined by origin_addr, origin_count and			
46 47	origin_da	tatype) to the buffer sp	ecified by argumen	ts target_count and target_datatype, at			
47 48	offset targ	get_disp, in the target w	indow specified by t	arget_rank and win, using the operation			

28

29 30

31

32

33 34

35

36

37

38

39

40

41

42 43

44

45

46 47

48

```
op. This is like MPI_PUT except that data is combined into the target area instead of overwriting it.
```

Any of the predefined operations for MPI_REDUCE can be used. User-defined functions cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both datatype arguments must be constructed from the same predefined datatype. The operation **op** applies to elements of that predefined type. The **parameter target_datatype** must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative function f(a,b) = b; i.e., the current value in the target memory is replaced by the value supplied by the origin.

MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE, MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not in collective reduction operations[,] such as MPI_REDUCE.

Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

Example 11.3 We want to compute $B(j) = \sum_{map(i)=j} A(i)$. The arrays A, B and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,
                                                                  &
                    comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i), m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL,
                                                               &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 11.2, page 441, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, then the

		444	CI	HAPTER 11. ONE-SIDED COMMUNICATIONS		
ticket270	$1 \\ 2 \\ 3 \\ \cdot \\ 4 \\ 5$	that previ	ious example.) In a similar m	s the reverse assignment to the one computed in anner, we can replace in Example 11.1, page 439, , thus performing the computation with only one ses.		
	6	Get Accun	nulate Function			
	7 8	It is often	n useful to have fetch-and-acc	cumulate semantics such that the remote data is		
	9	returned t	to the caller before the sent da	ata is accumulated into the remote data. The get		
	10		-	nically for each basic element in the datatype (see		
	11	behavior.	1.7 for details). The predefine	ed operation MPI_REPLACE provides fetch-and-set		
	12 13	benavior.				
	14					
	15	MPI_GET	_ACCUMULATE(origin_addr, result_count_result_datat	origin_count, origin_datatype, result_addr, type, target_rank, target_disp, target_count, target_datatype	2	
	16		op, win)		.,	
	17 18	IN	origin_addr	initial address of buffer (choice)		
	19	IN	origin_count	number of entries in origin buffer (non-negative inte-		
	20			ger)		
	21	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
	22 23	OUT	result_addr	initial address of result buffer (choice)		
	24 25	IN	result_count	number of entries in result buffer (non-negative inte- ger)		
	26	IN	result_datatype	datatype of each entry in result buffer (handle)		
	27 28	IN	target_rank	rank of target (non-negative integer)		
	28 29	IN	target_disp	displacement from start of window to beginning of tar-		
	30		taiBer-aibb	get buffer (non-negative integer)		
	31 32	IN	target_count	number of entries in target buffer (non-negative inte- ger)		
	33 34	IN	target_datatype	datatype of each entry in target buffer (handle)		
	35	IN	ор	reduce operation (handle)		
	36	IN	win	window object (handle)		
	37			(manal)		
ticket140a	38 	int MPI_	Get_accumulate(const void	<pre>*origin_addr, int origin_count,</pre>		
	40	<pre>MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype,</pre>				
	41					
	42	int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)				
ticket-248T	43 • 44					
	44 45	<pre>MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr, result_count, result_datatype, target_rank, target_disp,</pre>				
	46			_datatype, target_rank, target_disp, _datatype, op, win, ierror) BIND(C)		
	47	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr				
	48	TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr				

```
1
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                         \mathbf{2}
    target_count
                                                                                         3
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
    result_datatype
                                                                                         4
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                         5
                                                                                         6
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                         7
    TYPE(MPI_Win), INTENT(IN) ::
                                     win
                                                                                         8
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
                                                                                         9
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
                                                                                         10
               RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
                                                                                         11
               TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
                                                                                         12
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
                                                                                         13
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                         14
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
                                                                                         15
    TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
                                                                                         16
                                                                                         17
    Accumulate origin_count elements of type origin_datatype from the origin buffer (
                                                                                        18
origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank
                                                                                        19
and win, using the operation op and return in the result buffer result_addr the content of
                                                                                        20
the target buffer before the accumulation.
                                                                                        21
    The origin and result buffers (origin_addr and result_addr) must be disjoint. Each
                                                                                        22
datatype argument must be a predefined datatype or a derived datatype where all basic
                                                                                        23
components are of the same predefined datatype. All datatype arguments must be con-
                                                                                        24
structed from the same predefined datatype. The operation op applies to elements of that
                                                                                        25
predefined type. target_datatype must not specify overlapping entries, and the target buffer
                                                                                        26
must fit in the target window or in attached memory in a dynamic window. The operation
                                                                                        27
is executed atomically for each basic datatype; see Section 11.7 for details.
                                                                                        28
    Any of the predefined operations for MPI_REDUCE, and MPI_NO_OP or MPI_REPLACE
                                                                                        29
can be specified as op. User-defined functions cannot be used. A new predefined operation,
                                                                                        30
MPI_NO_OP, is defined. It corresponds to the associative function f(a,b) = a; i.e., the
                                                                                        31
current value in the target memory is returned in the result buffer at the origin and no
                                                                                         32
operation is performed on the target buffer. MPI_NO_OP can be used only in
                                                                                        33
MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP.
                                                                                        34
MPI_NO_OP cannot be used in MPI_ACCUMULATE, MPI_RACCUMULATE, or collective
                                                                                        35
reduction operations, such as MPI_REDUCE and others.
                                                                                        36
     Advice to users. MPI_GET is similar to MPI_GET_ACCUMULATE, with the opera-
                                                                                        37
     tion MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have
                                                                                        38
     different constraints on concurrent updates. (End of advice to users.)
                                                                                         39
                                                                                         40
                                                                                         41
Fetch and Op Function
                                                                                         42
The generic functionality of MPI_GET_ACCUMULATE might significantly limit the perfor-
                                                                                         43
mance of fetch-and-increment or fetch-and-add calls that might be supported by special
                                                                                         44
hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a
                                                                                         45
commonly used subset of the functionality of MPI_GET_ACCUMULATE.
                                                                                         46
```

47 48

	446		CHAPTER 11. ONE-SI	DED COMMUNICATIONS
1 2	MPI_FET	<pre>FCH_AND_OP(origin_</pre>	ddr, result_addr, datatype, targe	_rank, target_disp, op, win)
3	IN	origin_addr	initial address of buffer	(choice)
4 5	OUT	result_addr	initial address of result	buffer (choice)
6 7	IN	datatype	datatype of the entry in fers (handle)	n origin, result, and target buf-
8	IN	target_rank	rank of target (non-neg	ative integer)
10 11	IN	target_disp	displacement from start get buffer (non-negative	of window to beginning of tar- e integer)
12	IN	ор	reduce operation (hand	le)
13 14	IN	win	window object (handle)	r.
¹⁵ ticket140a. ¹⁶ 17 ticket-248T. ¹⁸	int MPI_Fetch_and_op(const void *origin_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win)			
19 20 21 22 23 24 25 26 27 28	<pre>MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,</pre>			
29 30 31 32 33 34 35 36 ticket270. ³⁷ 38 39 40 41 42 43	<typ INT INT Accor buffer at the opera before th The predefine specified</typ 	TARGET_DISP, pe> ORIGIN_ADDR(*) EGER(KIND=MPI_ADDRI EGER DATATYPE, TARC umulate one element offset target_disp, in ation op and return in he accumulation. origin and result buffe ed operations for MPI as op. User-defined f	DR, RESULT_ADDR, DATATYPE, OP, WIN, IERROR) RESULT_ADDR(*) SS_KIND) TARGET_DISP ET_RANK, OP, WIN, IERROR of type datatype from the origin the target window specified by the result buffer result_addr the rs (origin_addr and result_addr) n REDUCE, as well as MPI_NO_O unctions cannot be used. The data ation is executed atomically.	buffer (origin_addr) to the target_rank and win, using content of the target buffer must be disjoint. Any of the P or MPI_REPLACE, can be
44 45 46 ticket270. 47 48	Another compare		atomic compare and swap when arget, which is atomically repla	_

Section 11.7.1.]

MPI_CC	OMPARE_AND_SWAP(or target_disp, win)	rigin_addr, compare_addr, result_addr, datatype, target_rank,	1 2
IN	origin_addr	initial address of buffer (choice)	3
IN	compare_addr	initial address of compare buffer (choice)	4 5
OUT	result_addr	initial address of result buffer (choice)	6
IN	datatype	datatype of the element in all buffers (handle)	7
IN	target_rank	rank of target (non-negative integer)	8
IN	target_disp	displacement from start of window to beginning of tar-	10
	th Ber-disp	get buffer (non-negative integer)	11
IN	win	window object (handle)	12
		v x x	13 14
int MPI	void *result_a	onst void *origin_addr, const void *compare_addr, addr, MPI_Datatype datatype, int target_rank, et_disp, MPI_Win win)	$_{15}^{15}$ ticket140a. $_{16}^{16}$ ticket140a.
TYI	npare_and_swap(origin target_rank, t	n_addr, compare_addr, result_addr, datatype, target_disp, win, ierror) BIND(C) , INTENT(IN), ASYNCHRONOUS :: origin_addr,	 ¹⁷ ticket-248T. ¹⁸ ¹⁹ ²⁰ ²¹
		, ASYNCHRONOUS :: result_addr	22
	• •	<pre>FENT(IN) :: datatype target repl</pre>	23 24
	TEGER, INTENT(IN) :: TEGER(KIND=MPI ADDRES	target_rank SS_KIND), INTENT(IN) :: target_disp	25
	PE(MPI_Win), INTENT(I	°	26
	TEGER, OPTIONAL, INTE		27
	אדא אד איי איי איי	J_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,	28
FIF I_COI		TARGET_DISP, WIN, IERROR)	29
<tv< td=""><td></td><td>COMPARE_ADDR(*), RESULT_ADDR(*)</td><td>30 31</td></tv<>		COMPARE_ADDR(*), RESULT_ADDR(*)	30 31
•	TEGER(KIND=MPI_ADDRES		32
INT	TEGER DATATYPE, TARGE	T_RANK, WIN, IERROR	33
$\mathbf{T}\mathbf{h}^{i}$	is function compares on	e element of type datatype in the compare buffer	34
	-	t offset target_disp in the target window specified by	35
		s the value at the target with the value in the origin buffer	36
-		r and the target buffer are identical. The original value at	37
-		affer result_addr. The parameter datatype must belong to	38
	0 0	f predefined datatypes: C integer, Fortran integer, Logical,	39 40
		n Section 5.9.2 on page 188[, or can be of type MPI_AINT or	40 ticketxx:5/11
	(+5E) []. The origin and r	result buffers (origin_addr and result_addr) must be disjoint.	$_{42}^{11}$ ticketxx/11/

⁴⁰ ticketxx:5/1
 ⁴¹ ticketxx/11/
 ⁴² ticket270.
 ⁴⁴
 ⁴⁵ ticket270.

47 48

Unofficial Draft for Comment Only

[Any of the predefined operations for MPI_REDUCE, and MPI_NO_OP or MPI_REPLACE

can be specified as **op**. User-defined functions cannot be used. The outcome of accumulate

operations with overlapping types of different sizes or target displacements is undefined, see

1 2

3

4

5

6

7

10

12

13

14

15

16

17

18

19

20 21 22

23

24

ticket309.

txx:5/11/11. 11

Request-based RMA communication operations allow the user to associate a request handle with the RMA operations and test or wait for the completion of these requests using the functions described in Section 3.7.3, page 58. Request-based RMA operations are only valid within a passive-target epoch.

Upon returning from a completion call in which an RMA operation completes, the MPI_ERROR field in the associated status object is set appropriately (see Section 3.2.5 on page 34). [The values of the MPI_SOURCE and MPI_TAG fields are undefined.]All other fields of status and the results of status query functions (e.g., MPI_GET_COUNT) are undefined. It is valid to mix different request types ([i.e.]e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The end of the epoch, or explicit bulk synchronization using MPI_WIN_FLUSH,

MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally.

MPI_RPUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

24			
25	IN	origin_addr	initial address of origin buffer (choice)
26	IN	origin_count	number of entries in origin buffer (non-negative inte-
27			ger)
28	IN	origin_datatype	datatype of each entry in origin buffer (handle)
29 30	IN	target_rank	rank of target (non-negative integer)
31	IN	target_disp	displacement from start of window to target buffer
32			(non-negative integer)
33	IN	target_count	number of entries in target buffer (non-negative inte-
34			ger)
35	IN	target_datatype	datatype of each entry in target buffer (handle)
36			
37	IN	win	window object used for communication (handle)
38	OUT	request	RMA request (handle)
39		•	
40			

ticket140a.	40	<pre>int MPI_Rput(const void *origin_addr, int origin_count,</pre>
	42	MPI_Datatype origin_datatype, int target_rank,
	43	MPI_Aint target_disp, int target_count,
	44	MPI_Datatype target_datatype, MPI_Win win,
ticket-248T.	45	MPI_Request *request)
UICKCU-2401	46	MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
	47	<pre>target_disp, target_count, target_datatype, win, request,</pre>
	48	ierror) BIND(C)

TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr	1
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	2
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	3
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	4
TYPE(MPI_Win), INTENT(IN) :: win	5
TYPE(MPI_Request), INTENT(OUT) :: request	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
	8
MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	9
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,	10
IERROR)	11
<type> ORIGIN_ADDR(*)</type>	12
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	13
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	14
TARGET_DATATYPE, WIN, REQUEST, IERROR	15
MPI_RPUT is similar to MPI_PUT (Section 11.3.1), except that it allocates a commu-	16
nication request object and associates it with the request handle (the argument request).	17
The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) in-	18
dicates that the sender is now free to update the locations in the origin buffer. It does	19
not indicate that the data is available at the target window. If remote completion is re-	20
quired, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK or	21
MPI WIN UNLOCK ALL can be used	22

MPI_WIN_UNLOCK_ALL can be used.

MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

OUT	origin_addr	initial address of origin buffer (choice)	27
IN	origin_count	number of entries in origin buffer (non-negative inte-	28 29
		ger)	30
IN	origin_datatype	datatype of each entry in origin buffer (handle)	31
IN	target_rank	rank of target (non-negative integer)	32
IN	target_disp	displacement from window start to the beginning of	33
IIN	target_uisp	the target buffer (non-negative integer)	34
			35
IN	target_count	number of entries in target buffer (non-negative inte-	36
		ger)	37
IN	target_datatype	datatype of each entry in target buffer (handle)	38
INI			39
IN	win	window object used for communication (handle)	40
OUT	request	RMA request (handle)	41
			42
int MDT	Prot(woid torigin add	r int origin count	43

23 24

25

26

1	MPI_Rget	t(origin_addr, origin_	count, origin_datatype, target_rank,		
2	- 0		rget_count, target_datatype, win, request,		
3		ierror) BIND(C)			
4			SYNCHRONOUS :: origin_addr		
5			rigin_count, target_rank, target_count		
6		• -	<pre>IT(IN) :: origin_datatype, target_datatype</pre>		
7 8			KIND), INTENT(IN) :: target_disp		
9		E(MPI_Win), INTENT(IN)			
10		E(MPI_Request), INTENT EGER, OPTIONAL, INTENT	-		
11					
12	MPI_RGE1		COUNT, ORIGIN_DATATYPE, TARGET_RANK,		
13			RGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,		
14	(h	IERROR)			
15	• -	pe> ORIGIN_ADDR(*) EGER(KIND=MPI_ADDRESS_	KIND) TARGET DISD		
16			GIN_DATATYPE, TARGET_RANK, TARGET_COUNT,		
17		GET_DATATYPE, WIN, REQ			
18 19					
20			GET (Section 11.3.2), except that it allocates a commu-		
21		· ·	ates it with the request handle (the argument request) completion. The completion of an MPI_RGET operation		
22					
23	indicates that the data is available in the origin buffer. If origin_addr points to memo attached to a window, then the data becomes available in the private copy of this window				
24					
25					
26	MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_ target_count, target_datatype, op, win, request)				
27			· ·		
28 29	IN	origin_addr	initial address of buffer (choice)		
30	IN	origin_count	number of entries in buffer (non-negative integer)		
31	IN	origin_datatype	datatype of each buffer entry (handle)		
32	IN	target_rank	rank of target (non-negative integer)		
33	IN	target_disp	displacement from start of window to beginning of tar-		
34 35			get buffer (non-negative integer)		
36	IN	target_count	number of entries in target buffer (non-negative inte-		
37			ger)		
38	IN	target_datatype	datatype of each entry in target buffer (handle)		
39	IN	ор	reduce operation (handle)		
40	IN	win	window object (handle)		
41					
42 43	OUT	request	RMA request (handle)		
ticket140a. 44	int MDT	Raccumulate (const voi	.d *origin_addr, int origin_count,		
45	IIIC PHI.		igin_datatype, int target_rank,		
46			_disp, int target_count,		
47					
48		MPI_Request *rec			

<pre>MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,</pre>	<pre> 1 2 ticket-248T. 3 4 5 6 7 8 9 10 11 12 13 </pre>
<pre>MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>	13 14 15 16 17 18 19 20
MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.	21 22 23 24 25 26

1	MPI_RGE	T_ACCUMULATE(origi	in_addr, origin_count, origin_datatype, result_addr,
2 3		result_count, resu op, win, request)	<pre>ult_datatype, target_rank, target_disp, target_count, target_datatype,</pre>
4	IN	origin_addr	initial address of buffer (choice)
5 6 7	IN	origin_count	number of entries in origin buffer (non-negative inte- ger)
8	IN	origin_datatype	datatype of each buffer entry (handle)
9	OUT	result_addr	initial address of result buffer (choice)
10 11 12	IN	result_count	number of entries in result buffer (non-negative inte- ger)
13	IN	result_datatype	datatype of each buffer entry (handle)
14	IN	target_rank	rank of target (non-negative integer)
15 16	IN	target_disp	displacement from start of window to beginning of tar-
17 18			get buffer (non-negative integer)
19	IN	target_count	number of entries in target buffer (non-negative inte- ger)
20 21	IN	target_datatype	datatype of each buffer entry (handle)
22	IN	ор	reduce operation (handle)
23	IN	win	window object (handle)
24 25	OUT	request	RMA request (handle)
ticket140a. $\frac{26}{27}$ 28 29 30 31 ticket-248T. 32	int MPI_	MPI_Datatype of int result_cou int target_rar	ast void *origin_addr, int origin_count, origin_datatype, void *result_addr, unt, MPI_Datatype result_datatype, nk, MPI_Aint target_disp, int target_count, target_datatype, MPI_Op op, MPI_Win win, request)
33 34	MPI_Rget	•	addr, origin_count, origin_datatype,
35			result_count, result_datatype, target_rank, target_count, target_datatype, op, win, request,
36		ierror) BIND((· · · · ·
37			<pre>INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>
38 39			ASYNCHRONOUS :: result_addr
40		GER, INTENT(IN) :: get_count	origin_count, result_count, target_rank,
41	<u> </u>		ENT(IN) :: origin_datatype, target_datatype,
42	resu	lt_datatype	
43 44			S_KIND), INTENT(IN) :: target_disp
44		C(MPI_Op), INTENT(IN C(MPI_Win), INTENT(I	•
46			N/ win NT(OUT) :: request
47 48		GER, OPTIONAL, INTE	-

MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
IERROR)
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR

MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

$_{15}$ ticket270.

11.4 Memory Model

The memory semantics of RMA are best understood by using the concept of public and private window copies. We assume that systems have a public memory region that is addressable by all processes (e.g., the shared memory in shared memory machines or the exposed main memory in distributed memory machines). In addition, most machines have fast private buffers (e.g., transparent caches or explicit communication buffers) local to each process where copies of data elements from the main memory can be stored for faster access. Such buffers are either coherent, i.e., all updates to main memory are reflected in all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory need to be synchronized and updated in all private copies explicitly. Coherent systems allow direct updates to remote memory without any participation of the remote side. Non-coherent systems, however, need to call RMA functions in order to reflect updates to the public window in their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two memory models called *RMA unified*, if public and private window are logically identical, and *RMA separate*, otherwise.

In the RMA separate model, there is only one instance of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A local store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.1.

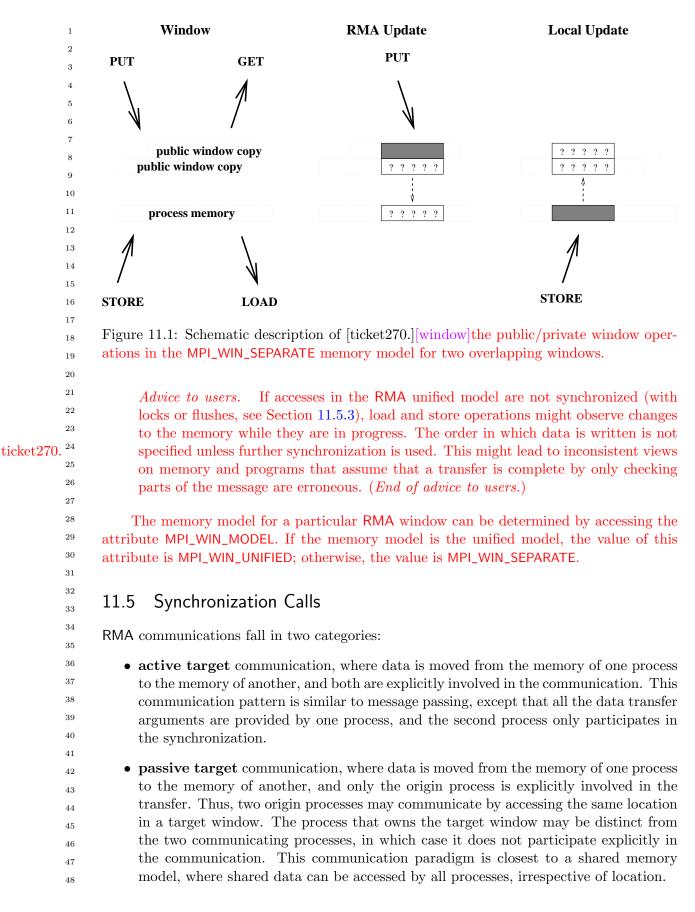
In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are observed by load operations without additional RMA calls. A store access to a window is visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance.

Unofficial Draft for Comment Only

 24

 31

 $_{36}$ ticket 284.



RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI_WIN_START and is terminated by a call to MPI_WIN_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.

48 ticket270.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

4

5

6 7

> 8 9

10

11

12

13

14

15

16

17

18

19 20

21

22

23

24

25

26

27

28

29 30

 31

32 33

34

35

36

37

38

39

40

41

42 43

44

 $45 \\ 46$

 3 ticket270.

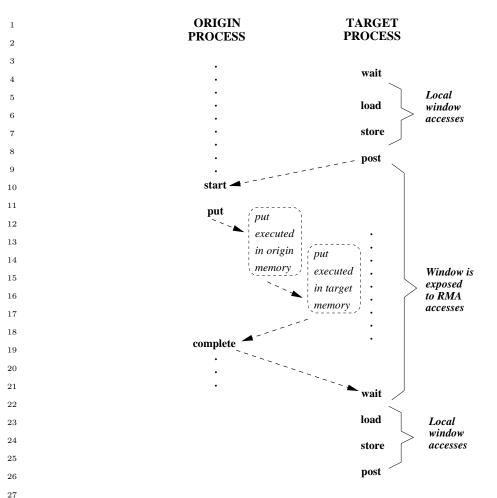


Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

3. [Finally, shared and exclusive locks are provided by the two functions MPI_WIN_LOCK and MPI_WIN_UNLOCK.]Finally, shared lock access is provided by the functions MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL. MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide ex-

clusive lock capability. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

These [two]four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively. [Only one target window can be accessed during that epoch with win.]

Figure 11.2 illustrates the general synchronization pattern for active target communication. The synchronization between **post** and **start** ensures that the put call of the origin process does not start until the target process exposes the window (with the **post** call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between **complete** and **wait** ensures that the put call of the origin process completes before the window is unexposed (with the **wait** call). The

Unofficial Draft for Comment Only

28

29 30

 31

32

33 34

35

36

37

ticket270.

ticket270.

ticket270

ticket270.

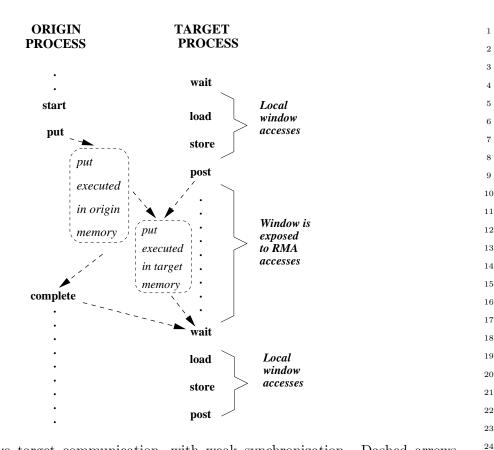


Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

target process will execute following local accesses to the target window only after the wait returned.

Figure 11.2 shows operations occurring in the natural temporal order implied by the synchronizations: the post occurs before the matching start, and complete occurs before the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as illustrated in Figure 11.3. The access to the target window is delayed until the window is exposed, after the post. However the start may complete earlier; the put and complete may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

Rationale. RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate[s], send/[recv]receive, etc.) needed to implement high-level synchronization operations. (*End of rationale.*)

Unofficial Draft for Comment Only

```
39
40
41
42
43 ticket270.
45
46 ticketxx:5/11
47 ticketxx:5/11
48
```

25

26 27

28

29

30

31

32

33

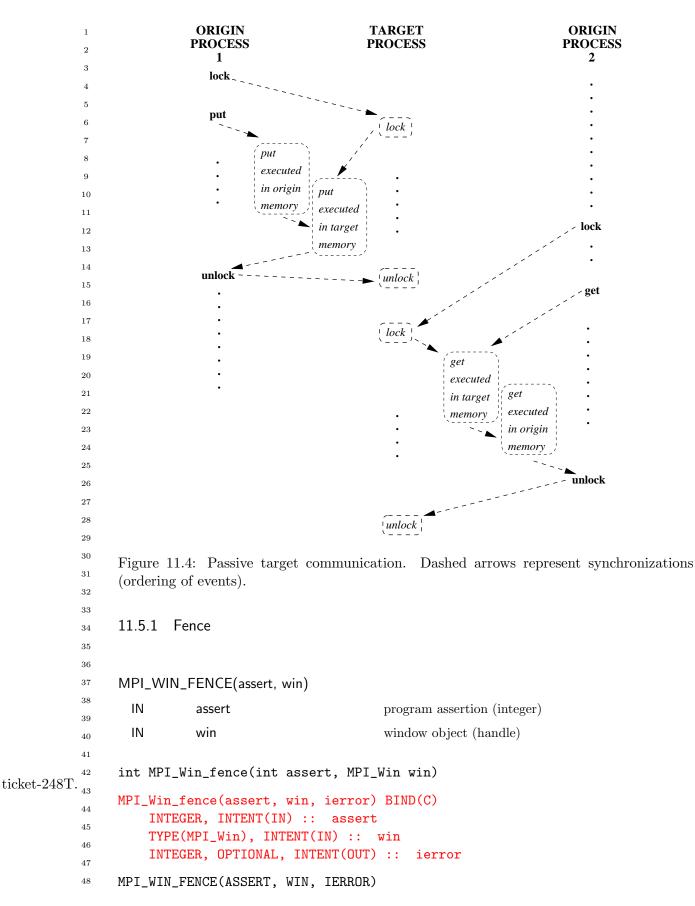
34

35

36

37

38



INTEGER	ASSERT,	WIN,	IERROR
---------	---------	------	--------

{void MPI::Win::Fence(int assert) const(binding deprecated, see Section 15.2) }

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and the local process issued RMA communication calls on win between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait.

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a call with $assert = MPI_MODE_NOPRECEDE$) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert =0 is always valid.

Calls to MPI_WIN_FENCE should both precede and follow calls Advice to users. to put, get or accumulate RMA communication functions that are synchronized with fence calls. (End of advice to users.)

11.5.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

	39
<pre>int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)</pre>	40 ticket-248T.
MPI_Win_start(group, assert, win, ierror) BIND(C)	41
TYPE(MPI_Group), INTENT(IN) :: group	42
INTEGER, INTENT(IN) :: assert	43
TYPE(MPI_Win), INTENT(IN) :: win	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
	46
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)	47
INTEGER GROUP, ASSERT, WIN, IERROR	48

1 $\mathbf{2}$

3 4

5

6

 $\overline{7}$

8

9

10

11

1213

14

15

16

17

18

19

20

21

22

23

24

25

26

2829 30

> 31 32 33

²⁷ ticket270.

		460 CHAPTER 11. ONE-SIDED COMMUNICATIONS
	1 2	<pre>{void MPI::Win::Start(const MPI::Group& group, int assert) const(binding</pre>
	3 4 5 6 7 8 9 10 11	Starts an RMA access epoch for win. RMA calls issued on win during this epoch must access only windows at processes in group. Each process in group must issue a matching call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary, until the target process executed the matching call to MPI_WIN_POST. MPI_WIN_START is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not required to. The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert = 0 is always unlid
	12 13	0 is always valid.
	14	MPI_WIN_COMPLETE(win)
	15 16	IN win window object (handle)
	17 18	int MPI_Win_complete(MPI_Win win)
ticket-248T.	19 20 21 22	MPI_Win_complete(win, ierror) BIND(C) TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	22 23 24	MPI_WIN_COMPLETE(WIN, IERROR) INTEGER WIN, IERROR
	25 26	<pre>{void MPI::Win::Complete() const(binding deprecated, see Section 15.2) }</pre>
	27 28 29 30 31 32 33	Completes an RMA access epoch on win started by a call to MPI_WIN_START. All RMA communication calls issued on win during this epoch will have completed at the origin when the call returns. MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin. Consider the sequence of calls in the example below.
ticket270.	34 · 35	Example 11.4
	36 37 38 39	<pre>MPI_Win_start(group, flag, win); MPI_Put(,win); MPI_Win_complete(win);</pre>
	40 41 42 43 44 45 46 47 48	The call to MPI_WIN_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process. This still leaves much choice to implementors. The call to MPI_WIN_START can block until the matching call to MPI_WIN_POST occurs at all target processes. One can also have implementations where the call to MPI_WIN_START is nonblocking, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurred; or implementations where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred; or even implementations where all three calls

can complete before any target process called MPI_WIN_POST — the data put must be 1 $\mathbf{2}$ buffered, in this last case, so as to allow the put to complete at the origin ahead of its 3 completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence 4above must complete, without further dependencies.

			5
			6
MPI_WI	N_POST(group, as	sert, win)	7
IN	group	group of origin processes (handle)	8
IN	assert	program assertion (integer)	9
			10
IN	win	window object (handle)	11
			12
int MP1	[_Win_post(MPI_G	roup group, int assert, MPI_Win win)	$^{13}_{14}$ ticket-248T.
MPI_Wir	n_post(group, as	sert, win, ierror) BIND(C)	
		NTENT(IN) :: group	15
	TEGER, INTENT(IN	U	16
TYF	PE(MPI_Win), INT	ENT(IN) :: win	17 18
INT	TEGER, OPTIONAL,	INTENT(OUT) :: ierror	19
MDT LITA		SERT, WIN, IERROR)	20
		ERT, WIN, IERROR	20
T 11 1	LUCEN GROOF, ASS.		22
{void M		onst MPI::Group& group, int assert) const(binding	23
	deprecated,	, see Section 15.2) }	24
Sta	rts an RMA exposu	re epoch for the local window associated with win. Only processes	25
	-	window with RMA calls on win during this epoch. Each process	26
• •		hing call to MPI_WIN_START. MPI_WIN_POST does not block.	27
0 1		· · · · · · · · · · · · · · · · · · ·	28
			29
MPI_WI	N_WAIT(win)		30
IN	win	window object (handle)	31
			32
int MP1	[_Win_wait(MPI_W	in win)	33
			34 ticket 229.1.
	n_wait(win, ierr		$_{35}$ ticket-248T.
	PE(MPI_Win), INT		36
TNI	IEGER, UPIIUNAL,	INTENT(OUT) :: ierror	37
MPI_WIN	J_WAIT(WIN, IERR	OR)	38
INT	TEGER WIN, IERRO	R	39 40
Swoid M	(DT··Win··Woit()	<pre>const(binding deprecated, see Section 15.2) }</pre>	40
l vora r	IIIWIIIWAIL()	consection 15.2)	42
	-	posure epoch started by a call to MPI_WIN_POST on win. This	43
		VIN_COMPLETE(win) issued by each of the origin processes that	44
were gra	anted access to the v	window during this epoch. The call to MPI_WIN_WAIT will block	45

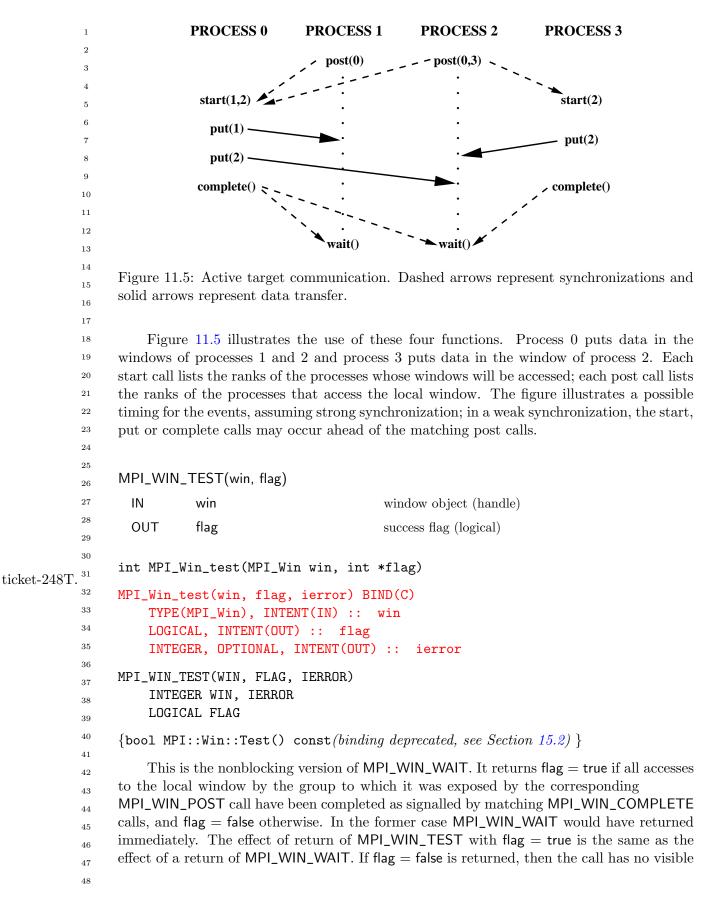
were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block 45until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all 46these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Unofficial Draft for Comment Only

 $\mathbf{5}$

47

48



effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

- MPI_WIN_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process in group, using wincomm. No need to wait for the completion of these sends.
- MPI_WIN_START(group,0,win) initiate a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.
- **MPI_WIN_COMPLETE(win)** initiate a nonblocking send with tag **tag1** to each process in the group of the preceding start call. No need to wait for the completion of these sends.
- **MPI_WIN_WAIT(win)** initiate a nonblocking receive with tag tag1 from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice[-] versa.

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs, in general: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph $G = \langle V, E \rangle$, where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process *i* accesses the window at target process *j*. Then each process *i* issues a call to MPI_WIN_POST(*ingroup*_i, ...), followed by a call to

MPI_WIN_START($outgroup_i,...$), where $outgroup_i = \{j : ij \in E\}$ and $ingroup_i = \{j : ji \in E\}$. A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

 $\mathbf{5}$

₂₄ ticket270.

CHAPTER 11. ONE-SIDED COMMUNICATIONS

```
1
                 11.5.3 Lock
            \mathbf{2}
            3
            4
                 MPI_WIN_LOCK(lock_type, rank, assert, win)
            5
                   IN
                             lock_type
                                                         either MPI_LOCK_EXCLUSIVE or
            6
                                                         MPI_LOCK_SHARED (state)
            7
            8
                   IN
                             rank
                                                         rank of locked window (non-negative integer)
            9
                   IN
                             assert
                                                         program assertion (integer)
            10
                                                         window object (handle)
                   IN
                             win
            11
            12
                 int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
            13
ticket-248T. _{14}
                 MPI_Win_lock(lock_type, rank, assert, win, ierror) BIND(C)
            15
                      INTEGER, INTENT(IN) :: lock_type, rank, assert
            16
                      TYPE(MPI_Win), INTENT(IN) :: win
            17
                      INTEGER, OPTIONAL, INTENT(OUT) ::
                                                             ierror
            18
                 MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
            19
                      INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
           20
           21
                 {void MPI::Win::Lock(int lock_type, int rank, int assert) const/binding
           22
                                 deprecated, see Section 15.2 }
           23
            ^{24}
                     Starts an RMA access epoch. Only the window at the process with rank rank can be
  ticket270. 25
                 accessed by RMA operations on win during that epoch.
            26
           27
                 MPI_WIN_LOCK_ALL(assert, win)
            28
                   IN
           29
                             assert
                                                         program assertion (integer)
            30
                   IN
                             win
                                                         window object (handle)
            ^{31}
            32
                 int MPI_Win_lock_all(int assert, MPI_Win win)
ticket-248T. ^{33}
            34
                 MPI_Win_lock_all(assert, win, ierror) BIND(C)
           35
                      INTEGER, INTENT(IN) :: assert
            36
                      TYPE(MPI_Win), INTENT(IN) :: win
           37
                      INTEGER, OPTIONAL, INTENT(OUT) ::
                                                              ierror
            38
                 MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
            39
                      INTEGER ASSERT, WIN, IERROR
            40
  ticket270. 41
                     Starts an RMA access epoch to all processes in win, with a lock type of
            42
                 MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on
            43
                 all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL
  ticket270. 44
                 must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL
  ticket270. 45
                 refers to a lock on all members of the group of the window.
            46
            47
                       Advice to users.
                                        There may be additional overheads associated with using
            48
                       MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These
```

overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when 1 $\mathbf{2}$ possible (see Section 11.5.5). (End of advice to users.)

			4
			5
MPI_WI	6		
IN	rank	rank of window (non-negative integer)	7
IN	win	window object (handle)	8
			9
int MPT	_Win_unlock(int rank, MPI_	Win win)	10
			11 ticket-248T.
	<pre>_unlock(rank, win, ierror)</pre>	BIND(C)	12
	EGER, INTENT(IN) :: rank		13
	E(MPI_Win), INTENT(IN) ::		14
INT	EGER, OPTIONAL, INTENT(OUT	[) :: ierror	15
MPI WIN	_UNLOCK(RANK, WIN, IERROR))	16
	EGER RANK, WIN, IERROR		17 18
			18
{void M	<pre>const(binding deprecated, see Section 15.2) }</pre>	20	
Con	npletes an RMA access epoch s	started by a call to MPI_WIN_LOCK(,win). RMA	20
operation	22		
when the	e call returns.		$_{23}$ ticket 270.
			24
			25
	N_UNLOCK_ALL(win)		26
IN	win	window object (handle)	27
			28
int MPI	_Win_unlock_all(MPI_Win wi	in)	29
MDT Win	_unlock_all(win, ierror) H		$_{30}$ ticket-248T.
	E(MPI_Win), INTENT(IN) ::		31
	EGER, OPTIONAL, INTENT(OUT		32
			33
MPI_WIN	34		
INTEGER WIN, IERROR			
Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL(assert, win). RMA operations issued during this epoch will have completed both at the origin and			

at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock calls, and to protect [local] load/store accesses to a locked local or shared memory window executed between the lock and unlock call. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. [I.e.]For example, a process may not call MPI_WIN_LOCK to lock a target window

Unofficial Draft for Comment Only

3

39

40

41

43

44

45

46

ticket270.

ticket284.

 $_{42}$ ticket 284.

 $_{48}$ ticket270.

	466	CHAPTER 11.	ONE-SIDED COMMUNICATIONS
1 2 3	if the target process has called MPI_V is erroneous to call MPI_WIN_POST		,
4 5 6 7 8 9 10	exposure epochs and locking p when locks or active target syn interactions between the two m here is that a set of windows	periods. But the achronization do echanisms. The p is used with only nechanism to and	to enforce mutual exclusion between is would entail additional overheads not interact in support of those rare programming style that we encourage y one synchronization mechanism at ther being rare and involving global
11 12 13 14			chronization code in order to enforce x posure epochs on a window. (<i>End of</i>
15 ticket270. 16 ticket270. 17 ticket270. 18	lock calls to windows in memory allo	ocated by MPI_A .2, page 425), o	munication that is synchronized by LLOC_MEM (Section 8.2, page 353), r attached with MPI_WIN_ATTACH y only in such memory.
20 ticket270. 21 ticket270. 22 $_{23}$ ticket270. 24 $_{25}$	shared [requires]may require ar implemented more easily, and ca allocated memory. It can be a	a asynchronous so an achieve better voided altogether that allows one t	t communication when memory is not oftware agent. Such an agent can be performance, if restricted to specially if shared memory is used. It seems to use shared memory for [3-rd]third es.
26 27 ticket270. 29 ticket270. 30 31 32	without taking advantage of no of C-like pointers; these are not dows/NT compilers, at the time	onstandard Fortr supported by so ne of writing)].	arget communication cannot be used an features: namely, the availability ome Fortran compilers[(g77 and Win- [Also, passive target communication or other statically declared Fortran
33 34	Consider the sequence of calls in	the example bel	ow.
35	Example 11.5		
36 37 38 39	<pre>MPI_Win_lock(MPI_LOCK_EXCLUSIVE MPI_Put(, rank,, win); MPI_Win_unlock(rank, win);</pre>	2, rank, assert	;, win);
40 41 42 43 44 45 46 47 ticket270. ⁴⁸	the origin and at the target. This st MPI_WIN_LOCK may block until an MPI_WIN_LOCK may not block, whi or, the first two calls may not block, w	ill leaves much fr exclusive lock or le the call to MPI thile MPI_WIN_U s then postponed VIN_LOCK is used , since the lock n	a the window is acquired; or, the call _PUT blocks until a lock is acquired; NLOCK blocks until a lock is acquired until the call to MPI_WIN_UNLOCK I to lock a local window, then the call
0101300210.	to the window issued after the IOCK C		

11.5.4	Flush and Sync		1	
All flusł	n and sync functions	can be called only within lock-unlock or lockall-unlockall epochs.	2 3	
			4	
MPI W	IN_FLUSH(rank, win	١	5	
		·	6	
IN	rank	rank of target window (non-negative integer)	7	
IN	win	window object (handle)	8	
			9	
int MP?	I_Win_flush(int ra	ank, MPI_Win win)	10 11 ticket-2487	
MPI_Wi	n_flush(rank, win	. ierror) BIND(C)	11 UICKet-240 I	
	TEGER, INTENT(IN)		12	
	PE(MPI_Win), INTE		14	
	TEGER, OPTIONAL,		15	
MDT LIT			16	
	N_FLUSH(RANK, WIN TEGER RANK, WIN, I		17	
			18	
		pletes all outstanding RMA operations initiated by the calling	19	
-		on the specified window. The operations are completed both at	20	
		t. Flush completes locally in the sense used in this document,	21	
		t return without requiring the target process to call any MPI	22	
routine.			23	
			24	
MPI_WIN_FLUSH_ALL(win)				
			26	
IN	win	window object (handle)	27	
			28	
int MP:	I_Win_flush_all(M	PI_Win win)	²⁹ ₃₀ ticket-2487	
MPT Wi	n_flush_all(win, :	ierror) RIND(C)		
	PE(MPI_Win), INTE		31	
		INTENT(OUT) :: ierror	32	
			33	
_	N_FLUSH_ALL(WIN, 1		34	
L N .	TEGER WIN, IERROR		35 36	
All	RMA operations issu	ued by the calling process to any target on the specified window	$^{36}_{37}$ ticket270.	
	-	specified window will have completed both at the origin and at	$_{38}^{37}$ ticket270.	
-		urns. MPI_WIN_FLUSH_ALL completes locally in the sense used	39	
_		that the call must return without requiring the target processes	40	
to call ε	any MPI routine.		41	
			42	
	IN_FLUSH_LOCAL(r		43	
			44	
IN	rank	rank of target window (non-negative integer)	45	
IN	win	window object (handle)	46	
			47	
			48	

ticket-248T.	1	int MPI	_Win_flush_l	ocal(int rank, MPI_Win win)			
01CRC0-2401.	-	MPI_Win	_flush_local	(rank, win, ierror) BIND(C)			
	4			C(IN) :: rank			
	5			INTENT(IN) :: win			
	6	TNI	EGER, OPTION	IAL, INTENT(OUT) :: ierror			
				.(RANK, WIN, IERROR)			
	8 9	INT	EGER RANK, W	/IN, IERROR			
	9 10	Locally completes at the origin all outstanding RMA operations initiated by the calling					
ticket 270.		process to the target process specified by rank on the specified window. For example, after					
		this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations. MPI_WIN_FLUSH_LOCAL completes locally in the sense used in this document,					
	10						
		routine.	; that the call	must return without requiring the target processes to call any MPI			
	10	routille.					
	16 17						
	18	MPI_WI	N_FLUSH_LO	CAL_ALL(win)			
	19	IN	win	window object (handle)			
	20						
ticket-248T.		int MPI	_Win_flush_l	.ocal_all(MPI_Win win)			
10100 - 10 1	22	MPI_Win	_flush_local	_all(win, ierror) BIND(C)			
	23 24			INTENT(IN) :: win			
	25	INT	EGER, OPTION	IAL, INTENT(OUT) :: ierror			
	26	MPI_WIN	_FLUSH_LOCAL	_ALL(WIN, IERROR)			
	27	INT	EGER WIN, IE	ERROR			
	28	All	RMA operatio	ons issued to any target prior to this call in this window will have			
	29 30	completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.					
	31			CAL_ALL completes locally in the sense used in this document, mean-			
	32	ing that	the call must r	return without requiring the target processes to call any MPI routine.			
	33						
	34	MPI_WI	N_SYNC(win)				
	35	IN	win	window object (handle)			
	36 37			• ()			
		int MPI	_Win_sync(MP	PI_Win win)			
ticket-248T.	39	MPT Win	sync(win i	error) BIND(C)			
	40		•	INTENT(IN) :: win			
	41			NAL, INTENT(OUT) :: ierror			
	42	MPT WTN	SYNC(WIN, I	FRROR)			
	43 44		EGER WIN, I				
4:-1	45						
ticket270.	46			N_SYNC synchronizes the private and public window copy of win. nchronizing the private and public window, MPI_WIN_SYNC has the			
	47	-	· · ·	eopening an access and exposure epoch on the window (note that it			
	48						

does not actually end an epoch or complete any pending [MPI]MPI RMA operations).	$^{1}_{2}$ ticket0.
11.5.5 Assertions	3
The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE[and], MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL is used to provide as- sertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the pro- gram — it is erroneous to provide[s] incorrect information. Users may always provide assert = 0 to indicate a general case[,] where no guarantees are made.	4 5 6 ticket270. 7 ticket270. 8 9 ticket270. 10 ticket270.
Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent[,] shared memory machines. Users should consult their implementation manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations[,] whenever available. (End of advice to users.)	$ \begin{array}{c} 11 \\ 12 \\ 13 \\ ticket270. \\ 14 \\ 15 \\ 16 \\ 17 \\ ticket270. \\ 10 \end{array} $
Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (End of advice to implementors.)	18 19 20 21
assert is the bit-vector OR of zero or more of the following integer constants: MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE and MPI_MODE_NOSUCCEED. The significant options are listed below[,] for each call.	$^{22}_{23}_{24}_{25}_{26}$ ticket270.
Advice to users. $C/C++$ users can use bit vector or () to combine these constants; Fortran 90 users can use the bit-vector IOR intrinsic. Fortran 77 users can use (non- portably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (End of advice to users.)	27 28 29 30 31 32
MPI_WIN_START:	33
MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already com- pleted on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)	34 35 36 37 38 39 40 41
MPI_WIN_POST:	42 43
MPI_MODE_NOCHECK — the matching calls to MPI_WIN_START have not yet oc- curred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.	44 45 46 47 48

	470 CHAPTER 11. ONE-SIDED COMMUNICATIONS	
1 2 3	MPI_MODE_NOSTORE — the local window was not updated by [local] stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.	ticket284.
4 5 6 7	MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.	
8	MPI_WIN_FENCE:	
ticket284. $_{10}^{9}$	MPI_MODE_NOSTORE — the local window was not updated by [local] stores (or local get or receive calls) since last synchronization.	
12 13	MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.	
$^{14}_{15}$ ticket270. $^{16}_{17}$	MPI_MODE_NOPRECEDE — the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.	
18 19 20	[MPI_MODE_NOSUCCEED]MPI_MODE_NOSUCCEED — the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.	
ticket270. $\frac{^{21}}{^{22}}$	MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:	
22 23 24 25 26	MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire a con- flicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.	
27 28 29 30	Advice to users. Note that the nostore and noprecede flags provide information on what happened <i>before</i> the call; the noput and nosucceed flags provide information on what will happen <i>after</i> the call. (<i>End of advice to users.</i>)	
31 32	11.5.6 Miscellaneous Clarifications	
33 34 35 36 37	Once an RMA routine completes, it is safe to free any opaque objects passed as argument to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete. As in message-passing, datatypes must be committed before they can be used in RMA	
ticket270. $\frac{1}{38}$	communication. [[Moved: Section on Examples]]	
40 41 42	11.6 Error Handling	
43	11.6.1 Error Handlers	
$\frac{\text{ticket 270.}}{\text{txx:}5/11/11.}_{46}^{45}$	Errors occurring during calls to [MPI_WIN_CREATE(,comm,)]routines that create MPI windows (e.g., MPI_WIN_CREATE(,comm,)) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.	

The default error handler associated with win is MPI_ERRORS_ARE_FATAL. Users may change this default by explicitly associating a new error handler with win (see Section 8.3, page 356).

11.6.2 Error Classes

The [following] error classes for one-sided communication are defined in Table 11.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP ticket270. ticket270. ticket270. ticket270. ticket270. ticket270.

		10
MPI_ERR_WIN	invalid win argument	11
MPI_ERR_BASE	invalid base argument	12
MPI_ERR_SIZE	invalid size argument	12
MPI_ERR_DISP	invalid disp argument	13
MPI_ERR_LOCKTYPE	invalid locktype argument	14
MPI_ERR_ASSERT	invalid assert argument	15
MPI_ERR_RMA_CONFLICT	conflicting accesses to window	
MPI_ERR_RMA_SYNC	[ticket270.][wrong]invalid synchronization of	$R_{18}^{17}MA$
	calls	18
[ticket270.]MPI_ERR_RMA_RANGE	[ticket270.]target memory is not part of the w	
	(in the case of a window created with	20
	MPI_WIN_CREATE_DYNAMIC, target memory	is_{not}^{21}
	attached)	22
[ticket270.]MPI_ERR_RMA_ATTACH	[ticket270.]memory cannot be attached (e.g., be	ecause
	of resource exhaustion)	24
[ticket284.]MPI_ERR_RMA_SHARED	[ticket284.]memory cannot be shared (e.g., som	e pro-
	cess in the group of the specified communicato	
	not expose shared memory)	21
[ticket284.]MPI_ERR_RMA_WRONG_FLAVOR	[ticket284.]passed window has the wrong flavor f	for the
	called function	20
		30
		31

Table 11.2: Error classes in one-sided communication routines

11.7 Semantics and Correctness

37 The semantics of RMA operations is best understood by assuming that the system main-38tains a separate *public* copy of each window, in addition to the original location in process 39 memory (the *private* window copy). There is only one instance of each variable in process 40 memory, but a distinct *public* copy of the variable for each window that contains it. A load 41 accesses the instance in process memory (this includes MPI sends). A store accesses and 42updates the instance in process memory (this includes MPI receives), but the update may 43affect other public copies of the same locations. A get on a window accesses the public copy 44of that window. A put or accumulate on a window accesses and updates the public copy of 45that window, but the update may affect the private copy of the same locations in process 46memory, and public copies of other overlapping windows. This is illustrated in Figure 11.1. 4748

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

 $\frac{4}{5}$

6

 36 ticket 270.

1 2 3 4 5 6 7	The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.
8 ticket270.9 10 11 12	 An RMA operation is completed at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_FENCE[or MPI_WIN_UNLOCK], MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the origin.
13 14 15 16	2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.
17 18 19 20	3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.
ticket270. 21 22 23 ticket270. 24	4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL, then the operation is completed at the target by that same call[to MPI_WIN_UNLOCK].
25 26 ticket270. 28 ticket270. 29 ticket270. 30 31	5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, [or MPI_WIN_UNLOCK]MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.
32 33 ticket270. ³⁴ ticket270. ³⁵ 36 37 38 39	6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE,[or MPI_WIN_LOCK]MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory without additional RMA calls.
40 41 42 43 ticket270. 44 ticket270. 45 46 ticket270. 47 ticket270. 48	The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. [On the other hand]In the RMA separate memory model, the update of private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed in the RMA separate memory model until the process executes a suitable synchronization call, while they have to complete in the RMA unified

ticket270.

model without additional synchronization calls. [Updates to a public window copy can also be delayed until the window owner executes a synchronization call, if fences or post-startcomplete-wait synchronization is used.]If fence or post-start-complete-wait synchronization is used, updates to a public window copy can be delayed in both memory models until the window owner executes a synchronization call. [Only when lock synchronization is used does it becomes necessary to update the public window copy, even if the window owner does not execute any related synchronization call.]When passive-target synchronization (lock/unlock or even flush) is used, it is necessary to update the public window copy in the RMA separate model, or the private window copy in the RMA unified model, even if the window owner does not execute any related synchronization call.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations may be *undefined* in some situations. For example, the result of several origin processes performing concurrent MPI_PUT operations to the same target location is undefined. In addition, the result of a single origin process performing multiple MPI_PUT operation to the same target location within the same access epoch is also undefined. The result at the target may have all of the data from one of the MPI_PUT operations (the "last" one, in some sense), or bytes from some of each of the operations, or something else. In MPI-2, such operations were *erroneous*. That meant that an MPI implementation was permitted to signal an MPI exception. Thus, user programs or tools that used MPI RMA could not portably permit such operations, even if the application code could function correctly with such an undefined result. In MPI-3, these operations are not erroneous, but do not have a defined behavior.

Rationale. As discussed in [6], requiring operations such as overlapping puts to be erroneous makes it [very]difficult to use MPI RMA to implement programming models—such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any implementation that enforces this rule, as it would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (*End of rationale.*)

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not generate an MPI exception. (*End of advice to implementors.*)

A [correct program]program with well-defined outcome in the MPI_WIN_SEPARATE memory model must obey the following rules.

- 1. A location in a window must not be accessed [locally]with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- 2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public

Unofficial Draft for Comment Only

473

1

 $\mathbf{2}$

3

4

6 7

8 9

10

11

12

13

14

15

17

18

19 20

21

22

23 24

25

26

27 28

29

30

31

32

33

34

35 36

37

38

39

40

42 43

44

45

 $46 \\ 47$

48

41 ticket270.

ticket284.

ticketxx:5/11

 16 ticket270.

⁵ ticket270.

1	
ticket270. 2	window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates [that use the same operation,]with the same
ticket270. ³	predefined datatype, on the same window. Additional restrictions on the operation
4	apply, see the info key accumulate_ops in Section 11.2.1.
5	
ticket284. $_{6}$	3. A put or accumulate must not access a target window once a [local]load/store update
ticket 270. $_7$	or a put or accumulate update to another (overlapping) target window [have]has started on a location in the target window, until the update becomes visible in the
ticket284. $_{9}^{8}$	public copy of the window. Conversely, a [local update in]store to process memory to
10	a location in a window must not start once a put or accumulate update to that target
10	window has started, until the put or accumulate update becomes visible in process
12	memory. In both cases, the restriction applies to operations even if they access disjoint
13	locations in the window.
ticket 270. $_{\scriptscriptstyle 14}$	
15	[A program is erroneous if it violates these rules.]
16	Rationale. The last constraint on correct RMA accesses may seem unduly restric-
17	tive, as it forbids concurrent accesses to nonoverlapping locations in a window. The
18 19	reason for this constraint is that, on some architectures, explicit coherence restoring
20	operations may be needed at synchronization points. A different operation may be
ticket284. $_{21}$	needed for locations that were [locally] updated by stores and for locations that were
22	remotely updated by put or accumulate operations. Without this constraint, the MPI
23	library will have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is
24	considered prohibitive. (<i>End of rationale.</i>)
ticket270. $^{25}_{26}$	
20	Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize
ticket270. $\frac{27}{28}$	the private and public window copies (that is, updates to one are made visible to the other).
29	In the MPI_WIN_UNIFIED memory model, the rules are much simpler because the public
30	and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with
31	a well-defined outcome must obey in this case are:
32	
ticket284. 33	1. A location in a window must not be accessed [locally] with load/store operations once
34 35	an update to that location has started, until the update is complete, subject to the
ticket 284. $_{36}$	following special case.
ticket284. 37	2. [Locally accessing (but not updating)]Accessing a location in the window [with a load
38	operation]that is also the target of a remote update is valid (not erroneous) but the
39	precise result will depend on the behavior of the implementation. Updates from a
40	remote process will appear in the memory of the target, but there are no atomicity or
41	ordering guarantees if more than one byte is updated. Updates are stable in the sense
42	that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero
43 44	or for a particular value, but not polling and comparing the relative magnitude of
45	values. Users are cautioned that polling on one memory location and then accessing a
46	different memory location has defined behavior only if the other rules given here and
47	in this chapter are followed.
48	

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (End of advice to users.)

- 3. [Locally u]Updating a location in the window with a store operation that is also the target of a remote read (but not update) is valid (not erroneous) but the precise result will depend on the behavior of the implementation. [Updates from the local process]Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits [the local process] to update memory [in its local window]with store operations without requiring a lock/unlock or other RMA synchronization epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and in this chapter are followed.
- 4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started until the update completes at the target. There is one exception to this rule: in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 11.2.1.
- 5. A put or accumulate must not access a target window once a [local update]store operation or a put or accumulate update to another (overlapping) target window has started on the same location in the target window until the update completes at the target window. Conversely, a [local update]store operation in process memory to a location in a window must not start once a put or accumulate update to the same location in that target window has started until the put or accumulate update completes at the target.

Note that MPI_WIN_FLUSH and MPI_WIN_FLUSH_ALL may be used within a passive target epoch to complete RMA operations at the target process. A program that violates these rules has undefined behavior.

Advice to users. A user can write correct programs by following the following rules:

- fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by [local] stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- **post-start-complete-wait:** A window should not be updated [locally]with store operations while being posted, if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location

1

 $\mathbf{2}$

3

4

5

6

7 ticket284.

⁸ ticket284.

	476	CHAPTER 11. ONE-SIDED COMMUNICATIONS
1 2	by accumulate calls) while the window is	. Locations accessed by get calls should not be updated posted.
3 4 5 6	that its window is n	ynchronization, the target process can tell the origin process ow ready for RMA access; with the complete-wait synchro- process can tell the target process that it has finished its window.
7 8 ticket284. 10	Nonconflicting acces	dow are protected by exclusive locks if they may conflict. ses (such as read-only accesses or accumulate accesses) are locks, both for [local]load/store accesses and for RMA ac-
12 13 14 15 16 17 ticket270. 18 19	mode, or change the lapping windows, wh to have the same va RMA accesses to th to MPI_WIN_WAIT, wait; after the call	Example 1 Constitution Constitu
20 21 22 23 24 25	operation is complete, and operation until that opera The RMA synchronization visible in public and priva	operations define when updates are guaranteed to become te windows. Updates may become visible earlier, but such
26 27 28	_	n dependent. (<i>End of advice to users.</i>) ed by the following examples:
ticket270. 29 30 31 32 33	inside a window for the separate and MPI_WIN_UNLOCK calls a	bllowing example demonstrates updating a memory location memory model, according to Rule 5. The MPI_WIN_LOCK round the store to X in process B are necessary to ensure and private copies of the window.
34 35	Process A:	Process B: window location X
36 37 38 39 40 41		<pre>MPI_Win_lock(EXCLUSIVE,B) store X /* local update to private copy of B */ MPI_Win_unlock(B) /* now visible in public window copy */</pre>
42 43	MPI_Barrier	MPI_Barrier
44 45 46 ticket270. 47 48	MPI_Win_lock(EXCLUSIVE,B) MPI_Get(X) /* ok, read fro MPI_Win_unlock(B)	n public window */

Example 11.7 In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining [local] load/stores and multi-process synchronization. Although the following example appears correct, the compiler or hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET returning the incorrect value of X.

Process A:	Process B: window location X	
	<pre>store X /* update to private&public copy of B */</pre>	
MPI_Barrier	MPI_Barrier	
MPI_Win_lock_all		
MPI_Get(X) /* ok, read fro	m window */	
MPI_Win_flush_local(B)		
/* read value in X */		
MPI_Win_unlock_all		

MPI_BARRIER provides process synchronization, but not [local] memory synchronization. The example could potentially be made safe through the use of compiler and hardware specific notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct result.

Example 11.8 [Rule 6:] The following example demonstrates the reading of a memory location updated by a remote process (Rule 6) in the RMA separate memory model. Although the MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is necessary to synchronize the private copy with the public copy.

Process A:	Process B: window location X
<pre>MPI_Win_lock(EXCLUSIVE,B) MPI_Put(X) /* update to pul MPI_Win_unlock(B)</pre>	olic window */
MPI_Barrier	MPI_Barrier
	<pre>MPI_Win_lock(EXCLUSIVE,B) /* now visible in private copy of B */ load X MPI_Win_unlock(B)</pre>

Note that in this example, the barrier is not critical to the semantic correctness. The use of exclusive locks guarantees a remote process will not modify the public copy after MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation looking for changes in X on process B would be semantically correct. The barrier is required to ensure that process A performs the put operation before process B performs the load of X.

Unofficial Draft for Comment Only

 $_{18}$ ticket 284.

 $_{24}$ ticket 270.

 48 ticket 270.

Example 11.9 Similar to Example 11.7, the following example is unsafe even in the unified
 model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. While
 Process B does not need to explicitly synchronize the public and private copies through
 MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the
 window, the scheduling of the load could result in old values of X being returned. Compiler
 and hardware specific notations could ensure the load occurs after the data is updated, or
 explicit one-sided synchronization calls can be used to ensure the proper result.

```
8
         9
               Process A:
                                              Process B:
         10
                                              window location X
         11
               MPI_Win_lock_all
               MPI_Put(X) /* update to window */
         12
               MPI_Win_flush(B)
         13
         14
               MPI_Barrier
                                              MPI_Barrier
         15
         16
                                              load X
         17
              MPI_Win_unlock_all
         18
         19
ticket270. 20
               Example 11.10 [The rules do not guarantee that process A in the following sequence will
               see the value of X as updated by the local store by B before the lock.]The following example
         21
               further clarifies Rule 5. MPI_WIN_LOCK and MPI_WIN_LOCK_ALL do not update the
         22
               public copy of a window with changes to the private copy. Therefore, there is no guarantee
         23
               that process A in the following sequence will see the value of X as updated by the local
         24
               store by process B before the lock.
         25
         26
               Process A:
                                              Process B:
         27
                                              window location X
         28
         29
                                              store X /* update to private copy of B */
         30
                                              MPI_Win_lock(SHARED,B)
         ^{31}
               MPI_Barrier
                                              MPI_Barrier
         32
         33
               MPI_Win_lock(SHARED,B)
         34
               MPI_Get(X) /* X may be the X before the store */
         35
               MPI_Win_unlock(B)
         36
                                              MPI_Win_unlock(B)
         37
                                              /* update on X now visible in public window */
         38
ticket270. 39
               The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would
         40
               guarantee process A would see the updated value of X, as the public copy of the window
         41
               would be explicitly synchronized with the private copy.
         42
ticket 270. ^{43}
               Example 11.11 [In the following sequence]Similar to the previous example, Rule 5 can have
         44
               unexpected implications for general active target synchronization with the RMA separate
         45
               memory model. It is not guaranteed that process B reads the value of X as per the local
         46
               update by process A, because neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by
         47
               process A ensure visibility in the public window copy.
         48
```

Process A:	Process B:	1
window location X		2
window location Y		3
		4
store Y		5
MPI_Win_post(A,B) /* Y vis	sible in public window */	6
MPI_Win_start(A)	MPI_Win_start(A)	7
		8
store X /* update to priva	ate window */	9
		10 11
MPI_Win_complete	MPI_Win_complete	11
MPI_Win_wait	·	12
/* update on x may not yet	t visible in public window */	14
MPI_Barrier	MPI_Barrier	15
hri_balliel	Mr1_Dalliel	16
	MPI_Win_lock(EXCLUSIVE,A)	17
	MPI_Get(X) /* may return an obsolete value */	18
	MPI_Get(Y)	19
	MPI_Win_unlock(A)	20
frank and a transformer		$_{21}$ ticket 270.
	ess B reads the value of X as per the local update by process	22
	WAIT nor MPI_WIN_COMPLETE calls by process A ensure	23
	copy.] To allow process B to read the value of X stored by	$_{24}$ ticket 270.
	laced by a local MPI_PUT that updates the public window ment X may become visible in the private copy [in]of process	25 ticle t 270
	MPI_WIN_WAIT call in process A. The update to Y made	$_{26}^{26}$ ticket270. $_{27}^{27}$ ticket270.
	Il is visible in the public window after the MPI_WIN_POST	
	otten by process B process B will read the proper value of	$^{28}_{29}$ ticket270.
	d be moved to the epoch started by the MPI_WIN_START	
	d still get the value stored by process A.	$^{30}_{31}$ ticket270.
	0 1	
Example 11.12 [Finally, in	the following sequence]The following example demonstrates	$^{32}_{_{33}}$ ticket270.
the interaction of general activ	e target synchronization with local read operations with the	34
	Rules 5 and 6 do <i>not</i> guarantee that the private copy of X	35
at process B has been updated	before the load takes place.	36
Process A:	Process B:	37
TIOCESS A.	window location X	38
		39
MPI_Win_lock(EXCLUSIVE,B)		40
MPI_Put(X) /* update to pu	ublic window */	41
MPI_Win_unlock(B)		42
		43
MPI_Barrier	MPI_Barrier	44
		45
	MPI_Win_post(B)	46
	MPI_Win_start(B)	47
		48

479

7

8

9

10 11

12

CHAPTER 11. ONE-SIDED COMMUNICATIONS

<pre>load X /* access to private window */ /* may return an obsolete value */</pre>
MPI_Win_complete MPI_Win_wait

ticket270. 6

[rules (5,6) do not guarantee that the private copy of X at B has been updated before the load takes place.] To ensure that the value put by process A is read, the local load must be replaced with a local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

11.7.1 Atomicity

13 The outcome of concurrent accumulate[ticket270.][s] operations to the same location[ticket270.][,] 14with the same [ticket270.][operation and] predefined datatype[ticket270.][,] is as if the accu-15mulates [ticket270.][where]were done at that location in some serial order. [ticket270.]Additional 16restrictions on the operation apply, see the info key accumulate_ops in Section 11.2.1. 17[ticket270.][On the other hand, if two locations are both updated by two accumulate calls, 18 then the updates may occur in reverse order at the two locations Concurrent accumulate 19 operations with different origin and target pairs are not ordered. Thus, there is no guaran-20tee that the entire call to [ticket270.][MPI_ACCUMULATE]an accumulate operation is exe-21cuted atomically. The effect of this lack of atomicity is limited: The previous correctness 22conditions imply that a location updated by a call to [ticket270.][MPI_ACCUMULATE,]an 23accumulate operation cannot be accessed by [ticket270.]a load or an RMA call other than 24 accumulate[ticket270.][,] until the [ticket270.][MPI_ACCUMULATE call]accumulate opera-25tion has completed (at the target). Different interleavings can lead to different results 26only to the extent that computer arithmetics are not truly associative or commutative. 27[ticket270.] The outcome of accumulate operations with overlapping types of different sizes 28or target displacements is undefined. 29

ticket270.

30

31

11.7.2 Ordering

32 Accumulate calls enable element-wise atomic read and write to remote memory locations. 33 MPI specifies ordering between accumulate operations from one process to the same (or 34overlapping) memory locations at another process on a per-datatype granularity. The de-35 fault ordering is strict ordering, which guarantees that overlapping updates from the same 36 source to a remote location are committed in program order and that reads (e.g., with 37 MPI_GET_ACCUMULATE) and writes (e.g., with MPI_ACCUMULATE) are executed and 38 committed in program order. Ordering only applies to operations originating at the same 39 origin that access overlapping target memory regions. MPI does not provide any guarantees 40for accesses or updates from different origins to overlapping target memory regions.

41 The default strict ordering may incur a significant performance penalty. MPI specifies 42the info key accumulate_ordering to allow relaxation of the ordering semantics when specified 43to any window creation function. The values for this key are as follows. If set to none, 44then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 45in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list of 46required access orderings at the target. Allowed values in the comma-separated list are rar, 47war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-after-write 48 ordering, respectively. These indicate whether operations of the specified type complete in

the order they were issued. For example, raw means that any writes must complete at the target before any reads. These ordering requirements apply only to operations issued by the same origin process and targeting the same target process. Note that rar, read-afterread, is included for completeness, as ordering is only important if an update (write) may be made.] The default value for accumulate_ordering is rar,raw,war,waw, which implies that writes complete at the target in the order in which they were issued, reads complete at the target before any writes that are issued after the reads, and writes complete at the target before any reads that are issued after the writes. Any subset of these four orderings can be specified. For example, if only read-after-read and write-after-write ordering is required, then the value of the accumulate_ordering key could be set to rar, waw. The order of values is not significant.

Note that the above ordering semantics apply only to accumulate operations, not puts and gets. Puts and gets within an epoch are unordered.

11.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled, then it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 11.4, on page 460. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5, on page 466. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 11.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock, if the order of the complete and wait calls is reversed, at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

4

5

6 $\overline{7}$

8 9

10

11

12

13

1415

16

17

19

20

21

22

23

 24

25

26

27

28

29

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

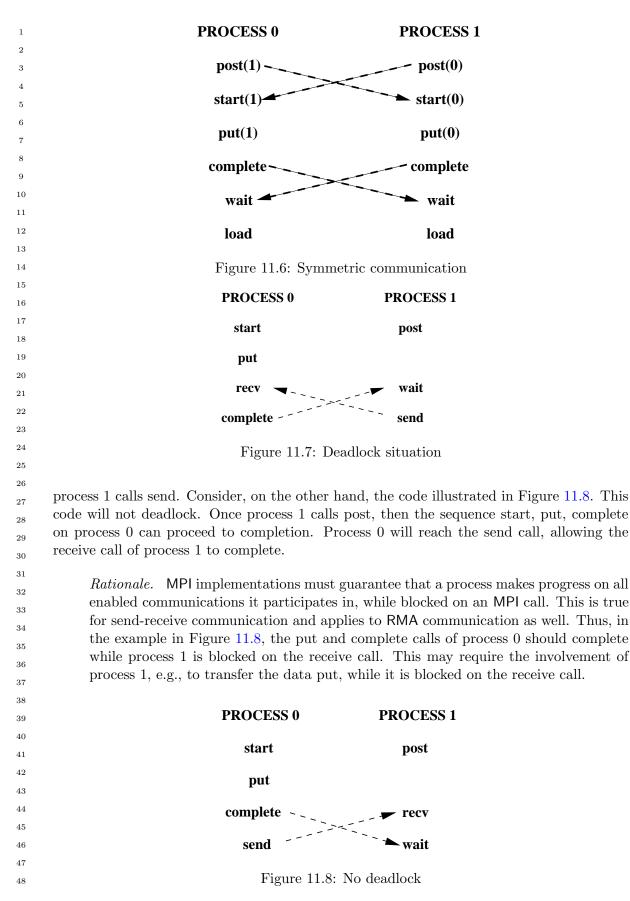
48

³ ticketxx:5/11

18 ticket270.

 $_{30}$ ticket 270.

 $_{47}$ ticket 270.



1 A similar issue is whether such progress must occur while a process is busy comput- $\mathbf{2}$ ing, or blocked in a non-MPI call. Suppose that in the last example the send-receive 3 pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not spec-4 ify whether deadlock is avoided. Suppose that the blocking receive of process 1 is $\mathbf{5}$ replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, 6 $\overline{7}$ even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait 8 9 call, or reaches another MPI call. The qualitative behavior is the same, under both 10 interpretations, unless a process is caught in an infinite compute loop, in which case 11 the difference may not matter. However, the quantitative expectations are different. 12Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI [f]Forum ¹³ ticket270. decided not to decide which interpretation of the standard is the correct one, since the 1415issue is very contentious, and a decision would have much impact on implementors 16but less impact on users. (End of rationale.)

11.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

The problem is illustrated by the following code:

Source of Process 1	Source of Process 2	Executed in Process 2	30
Source of Frocess 1	Source of Trocess 2	Executed in 1 locess 2	31
bbbb = 777	buff = 999	reg_A:=999	32
call MPI_WIN_FENCE	call MPI_WIN_FENCE		33
call MPI_PUT(bbbb		stop appl.thread	34
into buff of process 2)		buff:=777 in PUT handler	35
		continue appl.thread	36
call MPI_WIN_FENCE	call MPI_WIN_FENCE		37
	ccc = buff	ccc:=reg_A	38

In this example, variable **buff** is allocated in the register **reg_A** and therefore **ccc** will have the old value of **buff** and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 16.2.16.

[MPI implementations will avoid this problem for standard conforming C programs.]Programs4 written in C avoid this problem, because of the semantics of C. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in [COMMON blocks, or to variables that were 45 46 47 47 48 ticket238-J.

Unofficial Draft for Comment Only

17 18

19

20

21 22

23

24

25

27

28

29

39

40

41

42

 $_{26}$ ticket 270.

 $_{43}$ ticket 270.

1

 $\mathbf{2}$

 $\mathbf{5}$

 11 12

13

1415

16

20

21

22

23

33

41

ticket 270. 3

ticket270. 4

ticket270.⁶

ticket229.3. 7

ticket238-J. 8

ticket236-H. 9

ticket238-J.¹⁰

declared VOLATILE (while VOLATILE is not a standard Fortran declaration, it is supported by many Fortran compilers) (but this attribute may inhibit optimization of any code containing the RMA window). [Details]Further details and an additional solution are discussed in Section 16.2.16, "A Problem with Register Optimization," on page 679. See also[,] "Problems Due to Data Copying and Sequence Association," on page 673, for additional Fortran [problems] issues.] modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 673-676 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 16.2.16 to 16.2.19 on pages 679 to 688 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". Sections "Solutions" to "VOLATILE" on pages 682-686 discuss several solutions for the problem in this example.

11.8 Examples

ticket270. 17 This section was moved from earlier in the chapter. Changes and additions to this section 18 are marked in the same way as changes and additions in other parts of this chapter.] 19

> **Example 11.13** The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
^{24}
```

. . . while(!converged(A)){ 2526update(A); MPI_Win_fence(MPI_MODE_NOPRECEDE, win); 27for(i=0; i < toneighbors; i++)</pre> 28MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i], 29todisp[i], 1, totype[i], win); 30 MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win); 31 } 32

ticket270. 34 The same code could be written with get [] rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written 35 (as target buffer of puts). This is OK, provided that there is no overlap between the target 36 buffer of a put and another communication buffer. 37

38 **Example 11.14** Same generic example, with more computation/communication overlap. 39 We assume that the update phase is broken in two subphases: the first, where the "bound-40 ary," which is involved in communication, is updated, and the second, where the "core," which neither use nor provide communicated data, is updated. 42

43 . . . 44while(!converged(A)){ 45update_boundary(A); 46MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win); 47for(i=0; i < fromneighbors; i++)</pre> 48 MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],

ticket270.

```
fromdisp[i], 1, fromtype[i], win);
update_core(A);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
}
```

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the update_core call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

Example 11.15 Same code as in Example 11.13, rewritten using post-start-complete-wait.

Example 11.16 Same example, with split phases, as in Example 11.14.

Example 11.17 A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array A0 is updated using values of array A1, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window wini consists of array Ai.

 $\mathbf{2}$

```
1
              . . .
         \mathbf{2}
              if (!converged(A0,A1))
         3
                MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
         4
              MPI_Barrier(comm0);
              /* the barrier is needed because the start call inside the
         5
         6
              loop uses the nocheck option */
         7
              while(!converged(A0, A1)){
         8
                /* communication on AO and computation on A1 */
         9
                update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
         10
                MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
         11
                for(i=0; i < neighbors; i++)</pre>
         12
                   MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
         13
                               fromdisp0[i], 1, fromtype0[i], win0);
         14
                update1(A1); /* local update of A1 that is
         15
                                  concurrent with communication that updates A0 */
         16
                MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
         17
                MPI_Win_complete(win0);
         18
                MPI_Win_wait(win0);
         19
         20
                /* communication on A1 and computation on A0 */
         21
                update2(AO, A1); /* local update of AO that depends on A1 (and AO)*/
         22
                MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
         23
                for(i=0; i < neighbors; i++)</pre>
         24
                   MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                                fromdisp1[i], 1, fromtype1[i], win1);
         25
         26
                update1(A0); /* local update of A0 that depends on A0 only,
         27
                                 concurrent with communication that updates A1 */
         28
                if (!converged(A0,A1))
         29
                   MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
         30
                MPI_Win_complete(win1);
         31
                MPI_Win_wait(win1);
         32
                }
         33
                   A process posts the local window associated with win0 before it completes RMA accesses
         34
              to the remote windows associated with win1. When the wait(win1) call returns, then all
         35
              neighbors of the calling process have posted the windows associated with win0. Conversely,
         36
              when the wait(win0) call returns, then all neighbors of the calling process have posted the
         37
              windows associated with win1. Therefore, the nocheck option can be used with the calls to
         38
              MPI_WIN_START.
         39
                  Put calls can be used, instead of get calls, if the area of array A0 (resp. A1) used by
         40
              the update(A1, A0) (resp. update(A0, A1)) call is disjoint from the area modified by the
         41
              RMA communication. On some systems, a put call may be more efficient than a get call,
         42
ticket270.
              as it requires information exchange only in one direction.
         43
                  In the next several examples, for conciseness, the expression
         44
              z = MPI_Get_accumulate(...)
         45
         46
              means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr
         47
              in the description of MPI_GET_ACCUMULATE) on the left side of the assignment; in this
ticket270. 48
              case, z. This format is also used with MPI_COMPARE_AND_SWAP.
```

Example 11.18 The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE are used to write to or read from the local public copy.

•		7
Process A:	Process B:	8
MPI_Win_lock_all	MPI_Win_lock_all	9
window location X		10
X=2		11
MPI_Win_sync		12
MPI_Barrier	MPI_Barrier	13
		14
MPI_Accumulate(X, MPI_SUM, -1)	MPI_Accumulate(X, MPI_SUM, -1)	15
		16
stack variable z	stack variable z	17
do	do	18
<pre>z = MPI_Get_accumulate(X,</pre>	<pre>z = MPI_Get_accumulate(X,</pre>	19
MPI_NO_OP, 0)	MPI_NO_OP, 0)	20
MPI_Win_flush(A)	MPI_Win_flush(A)	21
while(z!=0)	while(z!=0)	22
		23
MPI_Win_unlock_all	MPI_Win_unlock_all	²⁴ +

Example 11.19 Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

		02
Process A:	Process B:	33
window location X	window location Y	34
window location T		35
		36
MPI_Win_lock_all	MPI_Win_lock_all	37
X=1	Y=1	38
MPI_Win_sync	MPI_Win_sync	39
MPI_Barrier	MPI_Barrier	40
<pre>MPI_Accumulate(T, MPI_REPLACE, 1)</pre>	<pre>MPI_Accumulate(T, MPI_REPLACE, 0)</pre>	41
stack variables t,y	<pre>stack variable t,x</pre>	42
t=1	t=0	43
y=MPI_Get_accumulate(Y,	<pre>x=MPI_Get_accumulate(X,</pre>	44
MPI_NO_OP, 0)	MPI_NO_OP, 0)	45
while(y==1 && t==1) do	while(x==1 && t==0) do	46
y=MPI_Get_accumulate(Y,	<pre>x=MPI_Get_accumulate(X,</pre>	47
MPI_NO_OP, O)	MPI_NO_OP, 0)	48

1 2

3 4

5

6

```
ticket 270.
```

25 26

27

 28

29

30

 31

```
1
                 t=MPI_Get_accumulate(T,
                                                            t=MPI_Get_accumulate(T,
         \mathbf{2}
                    MPI_NO_OP, 0)
                                                               MPI_NO_OP, 0)
         3
                MPI_Win_flush_all
                                                            MPI_Win_flush(A)
         4
               done
                                                          done
         \mathbf{5}
               // critical region
                                                         // critical region
         6
              MPI_Accumulate(X, MPI_REPLACE, 0)
                                                         MPI_Accumulate(Y, MPI_REPLACE, 0)
         7
               MPI_Win_unlock_all
                                                         MPI_Win_unlock_all
ticket270. <sub>8</sub>
         9
               Example 11.20 Implementing a critical region between multiple processes with compare
         10
               and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization
         11
               of A to guarantee the public copy has been updated with the initialization value found in
         12
               the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to
         13
               directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure
         14
               A in the public copy of Process A had been updated before the barrier.
         15
         16
               Process A:
                                                           Process B...:
         17
              MPI_Win_lock_all
                                                          MPI_Win_lock_all
         18
               atomic location A
         19
               A=0
         20
              MPI_Win_sync
         21
              MPI_Barrier
                                                          MPI_Barrier
               stack variable r=1
         22
                                                           stack variable r=1
         23
               while(r != 0) do
                                                           while(r != 0) do
                 r = MPI_Compare_and_swap(A, 0, 1)
         ^{24}
                                                             r = MPI_Compare_and_swap(A, 0, 1)
         25
                MPI_Win_flush(A)
                                                             MPI_Win_flush(A)
         26
               done
                                                           done
               // critical region
         27
                                                          // critical region
         28
              r = MPI_Compare_and_swap(A, 1, 0)
                                                          r = MPI_Compare_and_swap(A, 1, 0)
         29
              MPI_Win_unlock_all
                                                          MPI_Win_unlock_all
ticket270. 30
         31
               Example 11.21 The following example shows how request-based operations can be used
         32
               to overlap communication with computation. Each process fetches, processes, and writes
         33
               the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used
         34
               to allow up to M communication operations to overlap with computation.
         35
         36
               int
                            i, j;
         37
              MPI_Win
                            win;
         38
              MPI_Request put_req[M] = { MPI_REQUEST_NULL };
         39
              MPI_Request get_req;
         40
               double
                            **baseptr;
         41
                            data[M][N];
               double
         42
         43
              MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
         44
                MPI_COMM_WORLD, baseptr, &win);
         45
         46
              MPI_Win_lock_all(0, win);
         47
         48
              for (i = 0; i < NSTEPS; i++) {</pre>
```

488

```
MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
MPI_Win_unlock_all(win);
```

Example 11.22

The following example constructs a distributed shared linked list using dynamic windows. Initially process 0 creates the head of the list, attaches it to the window, and broadcasts the pointer to all processes. All processes then concurrently append N new elements to the list. When a process attempts to attach its element to the tail of the list it may discover that its tail pointer is stale and it must chase ahead to the new tail before the element can be attached. This example requires some modification to work in an environment where the length of a pointer is different on different processes.

```
#define NUM_ELEMS 10
/* Linked list pointer */
typedef struct {
 MPI_Aint disp;
  int
          rank;
} llist_ptr_t;
/* Linked list element */
typedef struct {
 llist_ptr_t next;
  int value;
} llist_elem_t;
const llist_ptr_t nil = { -1, (MPI_Aint) MPI_BOTTOM };
/* List of locally allocated list elements. */
static llist_elem_t **my_elems = NULL;
static int my_elems_size = 0;
static int my_elems_count = 0;
```

1

 $\mathbf{2}$

3

4 5 6

7

8

9 10

11

12 13 14

15

17

18

19

20

21

23

25 26 27

28 29

30

31

32

33

34 35

36

37

38

39

40 41

42 43

44

45

46

47 48

 16 ticket 270.

 $_{22}$ ticketxx:5/11

 $_{24}$ ticketxx:5/11

```
1
     /* Allocate a new shared linked list element */
\mathbf{2}
     MPI_Aint alloc_elem(int value, MPI_Win win) {
3
       MPI_Aint disp;
4
       llist_elem_t *elem_ptr;
5
6
       /* Allocate the new element and register it with the window */
7
       MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
8
       elem_ptr->value = value;
9
       elem_ptr->next = nil;
10
       MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
11
12
       /* Add the element to the list of local elements so we can free
13
          it later. */
14
       if (my_elems_size == my_elems_count) {
15
         my_elems_size += 100;
16
         my_elems = realloc(my_elems, my_elems_size);
17
       }
18
       my_elems[my_elems_count] = elem_ptr;
19
       my_elems_count++;
20
21
       MPI_Get_address(elem_ptr, &disp);
22
       return disp;
23
     }
^{24}
25
     int main(int argc, char **argv) {
26
                     procid, nproc, i;
       int
27
       MPI_Win
                     llist_win;
28
       llist_ptr_t head_ptr, tail_ptr;
29
30
       MPI_Init(&argc, &argv);
^{31}
32
       MPI_Comm_rank(MPI_COMM_WORLD, &procid);
33
       MPI_Comm_size(MPI_COMM_WORLD, &nproc);
34
35
       MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
36
37
       /* Process 0 creates the head node */
38
       if (procid == 0)
39
         head_ptr.disp = alloc_elem(-1, llist_win);
40
41
       /* Broadcast the head pointer to everyone */
42
       head_ptr.rank = 0;
       MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
43
44
       tail_ptr = head_ptr;
45
46
       /* Lock the window for shared access to all targets */
47
       MPI_Win_lock_all(0, llist_win);
48
```

```
1
/* All processes concurrently append NUM_ELEMS elements to the list */
                                                                                  2
for (i = 0; i < NUM_ELEMS; i++) {</pre>
                                                                                  3
  llist_ptr_t new_elem_ptr;
  int success;
                                                                                 4
                                                                                 5
                                                                                 6
  /* Create a new list element and attach it to the window */
                                                                                 7
  new_elem_ptr.rank = procid;
                                                                                  8
  new_elem_ptr.disp = alloc_elem(procid, llist_win);
                                                                                 9
                                                                                 10
  /* Append the new node to the list. This might take multiple
                                                                                 11
     attempts if others have already appended and our tail pointer
     is stale. */
                                                                                 12
  do {
                                                                                 13
                                                                                 14
    llist_ptr_t next_tail_ptr = nil;
                                                                                 15
                                                                                 16
    MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
                                                                                 17
         (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
                                                                                 18
        (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.rank),
                                                                                 19
        llist_win);
                                                                                 20
                                                                                 21
    MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                 22
    success = (next_tail_ptr.rank == nil.rank);
                                                                                 23
                                                                                 24
    if (success) {
                                                                                 25
      MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
                                                                                 26
          (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.disp), 1,
          MPI_AINT, MPI_REPLACE, llist_win);
                                                                                 27
                                                                                 28
                                                                                 29
      MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                 30
      tail_ptr = new_elem_ptr;
                                                                                 31
    } else {
                                                                                 32
                                                                                 33
      /* Tail pointer is stale, fetch the displacement. May take
                                                                                 34
         multiple tries if it is being updated. */
      do {
                                                                                 35
        MPI_Get_accumulate( NULL, 0, MPI_AINT, &next_tail_ptr.disp,
                                                                                 36
                                                                                 37
            1, MPI_AINT, tail_ptr.rank,
                                                                                 38
             (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.disp),
                                                                                 39
            1, MPI_AINT, MPI_NO_OP, llist_win);
                                                                                 40
                                                                                 41
        MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                 42
      } while (next_tail_ptr.disp == nil.disp);
      tail_ptr = next_tail_ptr;
                                                                                 43
    7
                                                                                 44
                                                                                 45
  } while (!success);
}
                                                                                 46
                                                                                 47
                                                                                 48
MPI_Win_unlock_all(llist_win);
```

```
1
        MPI_Barrier( MPI_COMM_WORLD );
\mathbf{2}
3
        /* Free all the elements in the list */
4
        for ( ; my_elems_count > 0; my_elems_count--) {
\mathbf{5}
           MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
6
          MPI_Free_mem(my_elems[my_elems_count-1]);
\overline{7}
        }
8
        MPI_Win_free(&llist_win);
9
      . . .
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Chapter 12

External Interfaces

12.1 Introduction

This chapter begins with calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. This can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. [This is]This functionality is needed for generalized requests.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI. Although thread compliance is not required, the standard specifies how threads are to work if they are provided.

12.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI_WAIT or MPI_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

Rationale. It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is very difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

 $46 \\ 47$

 $^{19}_{20}$ ticket 0.

	3] 4 ; 5 j	For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI when the operation completes. This is done by making a call to MPI_GREQUEST_COMPLETE. MPI maintains the "completion" status of generalized requests. Any other request state has to be maintained by the user. A new generalized request is started with					
		MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)					
	11 12	IN	query_fn	callback function invoked when request status is queried (function)			
1	13 14	IN	free_fn	callback function invoked when request is freed (function)			
1	15 16 17	IN	cancel_fn	callback function invoked when request is cancelled (function)			
	18	IN	extra_state	extra state			
	19	OUT	request	generalized request (handle)			
2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	22 23 24 25	<pre>int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,</pre>					
26 27 28 29 30 31 32 33	27 28 29 30 31 32 33	<pre>MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>					
2	34 35 36 37 38 39	MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR) INTEGER REQUEST, IERROR EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE					
4 4 4 4		{static M]	query_fn, const M const MPI::Greque	<pre>cart(const MPI::Grequest::Query_function* MPI::Grequest::Free_function* free_fn, est::Cancel_function* cancel_fn, e)(binding deprecated, see Section 15.2) }</pre>			
4	46 47 48	MPI::	Grequest, which is a der	a generalized request belongs, in C++, to the class ived class of MPI::Request. It is of the same type as tran. (<i>End of advice to users.</i>)			

1 The call starts a generalized request and returns a handle to it in request. $\mathbf{2}$ The syntax and meaning of the callback functions are listed below. All callback func-3 tions are passed the extra_state argument that was associated with the request by the starting call MPI_GREQUEST_START[. This can]; extra_state can be used to maintain 4 ticket0. user-defined state for the request. 56 In C, the query function is 7 typedef int MPI_Grequest_query_function(void *extra_state, 8 MPI_Status *status); 9 ¹⁰ ticket230-B. in Fortran with the mpi_f08 module ¹¹ ticket-248T. ABSTRACT INTERFACE 12SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror) 13BIND(C) 14TYPE(MPI_Status) :: status 15INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 16INTEGER :: ierror 17 ¹⁸ ticket230-B. in Fortran with the mpi module and mpif.h 19 SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) 20INTEGER STATUS(MPI_STATUS_SIZE), IERROR 21INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 22 and in C++2324{typedef int MPI::Grequest::Query_function(void* extra_state, 25MPI:::Status& status); (binding deprecated, see Section 15.2)} 26 ticket0. [query_fn] The query_fn function computes the status that should be returned for the 27generalized request. The status also includes information about successful/unsuccessful 28cancellation of the request (result to be returned by MPI_TEST_CANCELLED). ²⁹ ticket0. [query_fn]The query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} 30 call that completed the generalized request associated with this callback. The callback 31function is also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is com-32 33 34 3536

plete when the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI will pass a valid status object to query_fn, and this status will be ignored upon return of the callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE is called on the request; it may be invoked several times for the same generalized request, e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn callback functions, one for each generalized request that is completed by the MPI call. The order of these invocations is not specified by MPI.

In C, the free function is

typedef int MPI_Grequest_free_function(void *extra_state);

in Fortran with the mpi_f08 module

37

38

39

40

41

42

43

44 45

46

⁴⁷ ticket230-B. ⁴⁸ ticket-248T.

	1 2 3 4	ABSTRACT INTERFACE SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C) INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state INTEGER :: ierror
ticket230-B.	5 6	[and]in Fortran with the mpi module and mpif.h
ticket230-B.	8 9	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR) INTEGER IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
	10 11	and in C++
	12 13	<pre>{typedef int MPI::Grequest::Free_function(void* extra_state); (binding</pre>
ticket0. ticket0.	15	[free_fn]The free_fn function is invoked to clean up user-allocated resources when the generalized request is freed.
	16 17 18 19 20	[free_fn]The free_fn callback is invoked by the MPI_{WAIT TEST}{ANY SOME ALL} call that completed the generalized request associated with this callback. free_fn is invoked after the call to query_fn for the same request. However, if the MPI call completed multiple generalized requests, the order in which free_fn callback functions are invoked is not specified
ticket0.	21 22 23	by MPI. [free_fn]The free_fn callback is also invoked for generalized requests that are freed by a call to MPI_REQUEST_FREE (no call to WAIT_{WAIT TEST}{ANY SOME ALL} will occur
	24 25 26 27	for such a request). In this case, the callback function will be called either in the MPI call MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), whichever happens last, i.e., in this case the actual freeing code is executed as soon as both calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request
	28 29	is not deallocated until after free_fn completes. Note that free_fn will be invoked only once per request by a correct program.
	30 31	Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer
	32 33 34 35	valid. However, user copies of this handle are valid until after free_fn completes since MPI does not deallocate the object until then. Since free_fn is not called until after MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after free_fn executes. At
	36 37	this point, user copies of the request handle no longer point to a valid request. MPI will not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to
ticket0.	38 39 40	avoid accessing this stale handle. This is a special case [where]in which MPI defers deallocating the object until a later time that is known by the user. (<i>End of advice to users.</i>)
	41 42	In C, the cancel function is
	43 44	<pre>typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);</pre>
ticket230-B. ticket-248T.		<pre>in Fortran with the mpi_f08 module ABSTRACT INTERFACE SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)</pre>
	48	BIND(C)

```
1
       INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                             \mathbf{2}
       LOGICAL :: complete
                                                                                             3
       INTEGER :: ierror
                                                                                             4
                                                                                              ticket230-B.
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
                                                                                             6
                                                                                             7
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE
                                                                                             9
                                                                                            10
and in C++
                                                                                            11
{typedef int MPI::Grequest::Cancel_function(void* extra_state,
                                                                                            12
               bool complete); (binding deprecated, see Section 15.2)}
                                                                                            13
                                                                                            _{14} ticket0.
    [cancel_fn] The cancel_fn function is invoked to start the cancelation of a generalized
                                                                                            15
                                                                                            16 ticket0.
request. It is called by MPI_CANCEL(request). MPI passes to the callback function
complete=true]complete=true to the callback function if MPI_GREQUEST_COMPLETE was
                                                                                            17
already called on the request, and complete=false otherwise.
                                                                                            18
    All callback functions return an error code. The code is passed back and dealt with as
                                                                                            19
appropriate for the error code by the MPI function that invoked the callback function. For
                                                                                            20
example, if error codes are returned then the error code returned by the callback function
                                                                                            21
will be returned by the MPI function that invoked the callback function. In the case of
                                                                                            22
an MPI_{WAIT|TEST}{ANY} call that invokes both query_fn and free_fn, the MPI call will
                                                                                            23
return the error code returned by the last callback, namely free_fn. If one or more of the
                                                                                            24
requests in a call to MPI_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return
                                                                                            25
MPI_ERR_IN_STATUS. In such a case, if the MPI call was passed an array of statuses, then
                                                                                            26
MPI will return in each of the statuses that correspond to a completed generalized request
                                                                                            27
the error code returned by the corresponding invocation of its free_fn callback function.
                                                                                            28
However, if the MPI function was passed MPI_STATUSES_IGNORE, then the individual error
                                                                                            29
codes returned by each callback functions will be lost.
                                                                                            30
                                                                                            31
     Advice to users. query_fn must not set the error field of status since query_fn may
                                                                                            32
     be called by MPI_WAIT or MPI_TEST, in which case the error field of status should
                                                                                            33
     not change. The MPI library knows the "context" in which query_fn is invoked and
                                                                                            34
     can decide correctly when to put in the error field of status the returned error code.
                                                                                            35
     (End of advice to users.)
                                                                                            36
                                                                                            37
                                                                                            38
MPI_GREQUEST_COMPLETE(request)
                                                                                            39
                                                                                            40
  INOUT
            request
                                        generalized request (handle)
                                                                                            41
                                                                                            42
int MPI_Grequest_complete(MPI_Request request)
                                                                                            43
                                                                                              ticket-248T.
                                                                                            44
MPI_Grequest_complete(request, ierror) BIND(C)
    TYPE(MPI_Request), INTENT(IN) :: request
                                                                                            45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            46
                                                                                            47
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
                                                                                            48
```

	498 CHAPTER 12. EXTERNAL INTERFACES
1	INTEGER REQUEST, IERROR
$\frac{2}{3}$	<pre>{void MPI::Grequest::Complete()(binding deprecated, see Section 15.2) }</pre>
4 5 6 7 8 9 10 11 12 13 14 15 16	The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section 2.4). A call to MPI_WAIT(request, status) will return and a call to MPI_TEST(request, flag, status) will return flag=true only after a call to MPI_GREQUEST_COMPLETE has declared that these operations are complete. MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI_TEST, MPI_REQUEST_FREE, or MPI_CANCEL still hold. For example, all these calls are supposed to be local and nonblocking. Therefore, the callback functions query_fn, free_fn, or cancel_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined
17 18 19 20 21	operations. Advice to implementors. A call to MPI_GREQUEST_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (End of advice to implementors.)
22 23 24	12.2.1 Examples
25 26 27 28	Example 12.1 This example shows the code for a user-defined reduce operation on an int using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.
29 30 31 32 33 34 35 36 37	<pre>typedef struct { MPI_Comm comm; int tag; int root; int valin; int *valout; MPI_Request request; } ARGS;</pre>
38 39 40 41 42 43 44	<pre>int myreduce(MPI_Comm comm, int tag, int root,</pre>
45 46	<pre>/* start request */ MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);</pre>
47 48	
40	<pre>args = (ARGS*)malloc(sizeof(ARGS));</pre>

```
1
   args->comm = comm;
                                                                                     \mathbf{2}
   args->tag = tag;
                                                                                     3
   args->root = root;
   args->valin = valin;
                                                                                     4
                                                                                     5
   args->valout = valout;
                                                                                     6
   args->request = *request;
                                                                                     7
   /* spawn thread to handle request */
   /* The availability of the pthread_create call is system dependent */
                                                                                     9
                                                                                     10
   pthread_create(&thread, NULL, reduce_thread, args);
                                                                                     11
   return MPI_SUCCESS;
                                                                                     12
}
                                                                                     13
                                                                                     14
                                                                                     15
/* thread code */
                                                                                     16
void* reduce_thread(void *ptr)
                                                                                     17
{
                                                                                     18
   int lchild, rchild, parent, lval, rval, val;
                                                                                     19
   MPI_Request req[2];
   ARGS *args;
                                                                                     20
                                                                                     21
                                                                                     22
   args = (ARGS*)ptr;
                                                                                     23
                                                                                     24
   /* compute left, right child and parent in tree; set
                                                                                     25
      to MPI_PROC_NULL if does not exist */
                                                                                     26
   /* code not shown */
                                                                                     27
   . . .
                                                                                     28
                                                                                     29
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
                                                                                     30
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                     31
                                                                                     32
   val = lval + args->valin + rval;
                                                                                     33
   MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
                                                                                     34
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
   MPI_Grequest_complete((args->request));
                                                                                     35
                                                                                     36
   free(ptr);
                                                                                     37
   return(NULL);
}
                                                                                     38
                                                                                     39
int query_fn(void *extra_state, MPI_Status *status)
                                                                                     40
                                                                                     41
{
                                                                                     42
   /* always send just one int */
   MPI_Status_set_elements(status, MPI_INT, 1);
                                                                                     43
                                                                                     44
   /* can never cancel so always true */
   MPI_Status_set_cancelled(status, 0);
                                                                                     45
                                                                                     46
   /* choose not to return a value for this */
                                                                                     47
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                     48
   /* tag has no meaning for this generalized request */
```

```
1
        status->MPI_TAG = MPI_UNDEFINED;
\mathbf{2}
        /* this generalized request never fails */
3
        return MPI_SUCCESS;
4
     }
5
6
\overline{7}
     int free_fn(void *extra_state)
8
     {
9
        /* this generalized request does not need to do any freeing */
10
        /* as a result it never fails here */
11
        return MPI_SUCCESS;
12
     }
13
14
15
     int cancel_fn(void *extra_state, int complete)
16
     {
17
        /* This generalized request does not support cancelling.
18
            Abort if not already done. If done then treat as if cancel failed.*/
19
        if (!complete) {
20
           fprintf(stderr,
21
                    "Cannot cancel generalized request - aborting program\n");
22
           MPI_Abort(MPI_COMM_WORLD, 99);
23
24
        return MPI_SUCCESS;
25
     }
26
27
```

12.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls [use]to use the same request [mechanism. This]mechanism, which allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful [value]values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

 $46 \\ 47$

28 29

30

 31

32

33

34

35

36

37

38

30

40

41

42

43

44

45

ticket0.

ticket0.

ticket0.

MPI_STA	TUS_SET_ELEMENTS(status,	datatype, count)	1		
INOUT	status	status with which to associate count (Status)	2 3		
IN	datatype	datatype associated with count (handle)	4		
IN	count	number of elements to associate with status (integer)	5		
	count	number of ciclinents to associate with status (megor)	6		
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,					
	8				
NDT OL L			9 ticket-248T.		
<pre>MPI_Status_set_elements(status, datatype, count, ierror) BIND(C) TYPE(MPI_Status), INTENT(INOUT) :: status TYPE(MPI_Datatype), INTENT(IN) :: datatype</pre>					
	INTEGER, INTENT(IN) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
אסד פידאיזי	MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)				
		E), DATATYPE, COUNT, IERROR	16		
			17		
{void MP:	18				
	$^{19}_{20}$ ticket265.				
			20		
	TUS_SET_ELEMENTS_X(statu	us datatura sount)	22		
	•	,	23		
INOUT	status	status with which to associate count (Status)	24		
IN	datatype	datatype associated with count (handle)	25		
IN	count	number of elements to associate with status (integer)	26		
			27		
int MPI_	Status_set_elements_x(MPI	_Status *status, MPI_Datatype datatype,	28 29		
	MPI_Count count)				
MPI Stat	us set elements x(status.	datatype, count, ierror) BIND(C)	³⁰ ticket-248T.		
	(MPI_Status), INTENT(INOU	· · ·	32		
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	33		
INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) :: count					
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	35		
MPI_STAT	US_SET_ELEMENTS_X(STATUS,	DATATYPE, COUNT, IERROR)	36		
INTE	GER STATUS (MPI_STATUS_SIZ	E), DATATYPE, IERROR	37 38		
INTEGER (KIND=MPI_COUNT_KIND) COUNT					
[This	call modifies These functions	modify the opaque part of status so that a call to	³⁹ ticket265.		
		EMENTS_X will return count. MPI_GET_COUNT	41 ticket265.		
will return a compatible value.					
A subsequent call to MPI_GET_COUNT(status, datatype, count) [or to],					
MPI_GET_ELEMENTS(status, datatype, count), or MPI_GET_ELEMENTS_X(status, datatype,					
$MPI_GET_ELEMENTS(status, datatype, count), or MPI_GET_ELEMENTS_X(status, datatype, _{48ticket265}).$					

	502	CHAPTER .	12. EXTERNAL INTERFACES		
1 ticket265. 2 3 4	count) must use a datatype argument that has the same type signature as the datatype a gument that was used in the call to MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X.				
ticket0. 5 6 ticket265. 7 ticket265. 8 9 10 11	ticket265. 7 ticket265. 8 Pationale. [This]The requirement of matching type signatures for these similar to the restriction that holds when count is set by a receive operate that case, the calls to MPI_GET_COUNT[and], MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X must use a datatype with the same signature as the do used in the receive call. (End of rationale.)				
12 13	MPI_STATUS_SET_CANCELLED(status, flag)				
14	INOUT status	status with which	n to associate cancel flag (Status)		
15 16	IN flag	if true indicates r	request was cancelled (logical)		
¹⁷ ticket-248T. ¹⁸	<pre>int MPI_Status_set_cancelled(MPI_Status *status, int flag)</pre>				
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35	<pre>} If flag is set to true then a also return flag = true, other Advice to users. Use than those for which tl when using the status of</pre>	<pre>TENT(INOUT) :: status :: flag VTENT(OUT) :: ierror (STATUS, FLAG, IERROR) TATUS_SIZE), IERROR ancelled(bool flag)(bindi a subsequent call to MPI_TES wise it will return false. rs are advised not to reuse ney were intended. Doing so bject. For example, calling N</pre>	ing deprecated, see Section 15.2) ST_CANCELLED(status, flag) will the status fields for values other may lead to unexpected results MPI_GET_ELEMENTS may cause apossible to detect such an error.		
36 37 38 39 40 41	The extra_state argume information that does values in a status set in	nt provided with a generaliz not logically belong in state	ed request can be used to return us. Furthermore, modifying the RECV, may lead to unpredictable		
42	12.4 MPI and Threads	5			
43 44 45 46 47 48	minimal requirements for th that can be used for initiali environments where threads	read compliant MPI implex zing the thread environment are not supported or perfo	and threads. The section lists mentations and defines functions t. MPI may be implemented in orm poorly. Therefore, it is not rements specified in this section.		

This section generally assumes a thread package similar to POSIX threads [38], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

12.4.1 General

In a thread-compliant implementation, an MPI process is a process that may be multithreaded. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

Rationale. This model corresponds to the POSIX model of interprocess communication: the fact that a process is multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations [where]in which MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (*End of rationale.*)

Advice to users. It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (*End of advice to users.*)

The two main requirements for a thread-compliant implementation are listed below.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

35**Example 12.2** Process 0 consists of two threads. The first thread executes a blocking 36 send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes 37 a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first 38thread sends a message that is received by the second thread. This communication should 39 always succeed. According to the first requirement, the execution will correspond to some 40 interleaving of the two calls. According to the second requirement, a call can only block 41 the calling thread and cannot prevent progress of the other thread. If the send call went 42ahead of the receive call, then the sending thread may block, but this will not prevent 43the receiving thread from executing. Thus, the receive call will occur. Once both calls 44occur, the communication is enabled and both calls will complete. On the other hand, a 45single-threaded process that posts a send, followed by a matching receive, may deadlock. 46The progress requirement for multithreaded implementations is stronger, as a blocked call 47cannot prevent progress in other threads. 48

1

 $\mathbf{2}$

3

4 5

6

 $\overline{7}$

8

9

10 11

12

13

15

16 17

18

19

20

21 22

23

24 25

26

27

28

29

30

31

32

33 34

 $_{14}^{14}$ ticket0.

Advice to implementors. MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (*End of advice to implementors.*)

12.4.2 Clarifications

Initialization and Completion The call to MPI_FINALIZE should occur on the same thread
 that initialized MPI. We call this thread the main thread. The call should occur only after
 all the process threads have completed their MPI calls, and have no pending communications
 or I/O operations.

- 16
- 17 18

26

27

28

29

30

31

32 33 Rationale. This constraint simplifies implementation. (End of rationale.)

Multiple threads completing the same request. A program where two threads block, waiting
 on the same request, is erroneous. Similarly, the same request cannot appear in the array of
 requests of two concurrent MPI_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a request
 ticket0. 22
 can only be completed once. Any combination of wait or test [which]that violates this rule
 is erroneous.

ticket0. $^{24}_{25}$

Rationale. [This] This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (End of rationale.)

Probe A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process.

40

Collective calls Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

Advice to users. With three concurrent threads in each MPI process of a communicator comm, it is allowed that thread A in each MPI process calls a collective operation on comm, thread B calls a file operation on an existing filehandle that was formerly

1

2

3

4

5

6

7

8

9 10

opened on **comm**, and thread C invokes one-sided operations on an existing window handle that was also formerly created on **comm**. (*End of advice to users.*)

Rationale. As already specified in MPI_FILE_OPEN and MPI_WIN_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. [Advice to implementors.] If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (End of advice to implementors.)

Exception handlers An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that is distinct from the thread that will return the error code.

Rationale. The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

Rationale. Few C library functions are signal safe, and many have cancellation points — points [where]at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancelsafe" or ["async-signal-safe."] "async-signal-safe"). (End of rationale.)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

12.4.3 Initialization

The following function may be used to initialize MPI, and initialize the MPI thread environment, instead of MPI_INIT.

 $\mathbf{5}$

 24

 31

9 ticket0.

28 ticket0.

30 ticket0.

```
1
                 MPI_INIT_THREAD(required, provided)
            2
                   IN
                             required
                                                          desired level of thread support (integer)
            3
                   OUT
                             provided
                                                          provided level of thread support (integer)
            4
            5
            6
                 int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
                                 int *provided)
            7
ticket-248T.
            8
                 MPI_Init_thread(required, provided, ierror) BIND(C)
            9
                      INTEGER, INTENT(IN) :: required
            10
                      INTEGER, INTENT(OUT) :: provided
            11
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            12
                 MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
            13
            14
                      INTEGER REQUIRED, PROVIDED, IERROR
            15
                 {int MPI::Init_thread(int& argc, char**& argv, int required)(binding
            16
                                 deprecated, see Section 15.2 }
            17
            18
                 {int MPI:::Init_thread(int required)(binding deprecated, see Section 15.2) }
            19
            20
                       Advice to users. In C and C++, the passing of argc and argv is [optional.] optional,
    ticket0.
            21
    ticket0.
                       as with MPI_INIT as discussed in Section 8.7. In C, [this is accomplished by passing
            22
                       the appropriate null pointer.] null pointers may be passed in their place. In C++,
            23
                       [this is accomplished with two separate bindings to cover these two cases. This is as
    ticket0.
            24
                       with MPI_INIT as discussed in Section 8.7.] two separate bindings support this choice.
            25
                       (End of advice to users.)
            26
           27
                      This call initializes MPI in the same way that a call to MPI_INIT would. In addition,
            28
                 it initializes the thread environment. The argument required is used to specify the desired
            29
                 level of thread support. The possible values are listed in increasing order of thread support.
            30
                 MPI_THREAD_SINGLE Only one thread will execute.
           31
            32
                 MPI_THREAD_FUNNELED The process may be multi-threaded, but the application must
            33
                       ensure that only the main thread makes MPI calls (for the definition of main thread,
            34
                       see MPI_IS_THREAD_MAIN on page 508).
           35
                 MPI_THREAD_SERIALIZED The process may be multi-threaded, and multiple threads may
           36
                       make MPI calls, but only one at a time: MPI calls are not made concurrently from
           37
                       two distinct threads (all MPI calls are "serialized").
            38
            39
                 MPI_THREAD_MULTIPLE Multiple threads may call MPI, with no restrictions.
            40
           41
                 These values are monotonic; i.e., MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED <
           42
                 MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE.
            43
                      Different processes in MPI_COMM_WORLD may require different levels of thread sup-
            44
                 port.
            45
                      The call returns in provided information about the actual level of thread support that
            46
                 will be provided by MPI. It can be one of the four values listed above.
            47
                     The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend
            48
                 on the implementation, and may depend on information provided by the user before the
```

program started to execute (e.g., with arguments to mpiexec). If possible, the call will return provided = required. Failing this, the call will return the least supported level such that provided > required (thus providing a stronger level of support than required by the user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

A thread compliant MPI implementation will be able to return provided = MPI_THREAD_MULTIPLE. Such an implementation may always return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required. At the other extreme, an MPI library that is not thread compliant may always return provided = MPI_THREAD_SINGLE, irrespective of the value of required.

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required = MPI_THREAD_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of thread support available when the MPI program is started, e.g., with arguments to mpiexec. This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required; a call to MPI_INIT will also initialize the MPI thread support level to MPI_THREAD_MULTIPLE. Suppose, on the other hand, that an MPI program has been started so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will return provided = required; on the other hand, a call to MPI_INIT will initialize the MPI thread support level to MPI_THREAD_SINGLE.

Rationale. Various optimizations are possible when MPI code is executed singlethreaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits many applications well, e.g., if the process code is a sequential Fortran/C/C++ program with MPI calls that has been parallelized by a compiler for execution on an SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but MPI calls will likely execute on a single thread.

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C/ C++/Fortran library calls that are not thread safe, e.g., in an environment where malloc is not thread safe, then malloc should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5

6 7

8 9

10 11

12

13

14 15

16

17

18

19

20

21

22 23

 24

25

26

27

28

29

30

 31

32 33

34

35

36 37

38

39

40

41

42

43

44

45

 $46 \\ 47$

1 MPI_QUERY_THREAD(provided) 2 OUT provided provided level of thread support (integer) 3 4 int MPI_Query_thread(int *provided) 5ticket-248T. 6 MPI_Query_thread(provided, ierror) BIND(C) 7 INTEGER, INTENT(OUT) :: provided 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 MPI_QUERY_THREAD(PROVIDED, IERROR) 10 INTEGER PROVIDED, IERROR 11 12{int MPI::Query_thread()(binding deprecated, see Section 15.2) } 13 ticket0. 14 The call returns in provided the current level of thread [support. This]support, which will be the value returned in provided by MPI_INIT_THREAD, if MPI was initialized by a 15 call to MPI_INIT_THREAD(). 161718 MPI_IS_THREAD_MAIN(flag) 19OUT flag true if calling thread is main thread, false otherwise 20(logical) 2122 23int MPI_Is_thread_main(int *flag) ticket-248T. 24 MPI_Is_thread_main(flag, ierror) BIND(C) 25LOGICAL, INTENT(OUT) :: flag 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728MPI_IS_THREAD_MAIN(FLAG, IERROR) 29 LOGICAL FLAG 30 INTEGER IERROR 31 {bool MPI::Is_thread_main() (binding deprecated, see Section 15.2) } 32 ticket0. ³³ This function can be called by a thread to [find out whether] determine if it is the main 34thread (the thread that called MPI_INIT or MPI_INIT_THREAD). 35 All routines listed in this section must be supported by all MPI implementations. 36 MPI libraries are required to provide these calls even if they do not 37 Rationale. support threads, so that portable code that contains invocations to these functions 38 ticket0. 39 be able to can link correctly. MPI_INIT continues to be supported so as to provide compatibility with current MPI codes. (End of rationale.) 4041 It is possible to spawn threads before MPI is initialized, but no Advice to users. 42MPI call other than MPI_INITIALIZED should be executed by these threads, until 43 MPI_INIT_THREAD is invoked by one thread (which, thereby, becomes the main 44 thread). In particular, it is possible to enter the MPI execution with a multi-threaded 45process. 4647 The level of thread support provided is a global property of the MPI process that can 48 be specified only once, when MPI is initialized on that process (or before). Portable

Unofficial Draft for Comment Only

Unofficial Draft for Comment Only

Chapter 13

I/O

13.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [47], collective buffering [7, 14, 48, 52, 59], and disk-directed I/O [43]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

13.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 24

filetype A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be non-negative and monotonically nondecreasing.

view A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 13.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

etype
filetype holes
tiling a file with the filetype: displacement accessible data
Figure 13.1: Etypes and filetypes
A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure 13.2).
etype
process 0 filetype
process 1 filetype
process 2 filetype
tiling a file with the filetypes:
Figure 13.2: Partitioning a file among parallel processes

offset An offset is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure 13.2is the position of the 8th etype in the file after the displacement. An "explicit offset" is an offset that is used as a formal parameter in explicit data access routines.

Unofficial Draft for Comment Only

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

13.2 File Manipulation

13.2.1 Opening a File

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)	21
			22
IN	filename	name of file to open (string)	23
IN	amode	file access mode (integer)	24
IN	info	info object (handle)	25
OUT	fh		26
OUT	fh	new file handle (handle)	27

- int MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)
- MPI_File_open(comm, filename, amode, info, fh, ierror) BIND(C)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 CHARACTER(LEN=*), INTENT(IN) :: filename
 INTEGER, INTENT(IN) :: amode
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(MPI_File), INTENT(OUT) :: fh
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) CHARACTER*(*) FILENAME INTEGER COMM, AMODE, INFO, FH, IERROR

MPI_FILE_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI_FILE_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference

Unofficial Draft for Comment Only

29 ticket140.

ticket-248T.

1 the same file. (Values for info may vary.) comm must be an intracommunicator; it is erroneous to pass an intercommunicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN 3 are raised using the default file error handler (see Section 13.7, page 576). A process can 4 open a file independently of other processes by using the MPI_COMM_SELF communicator. $\mathbf{5}$ The file handle returned, fh, can be subsequently used to access the file until the file is 6 closed using MPI_FILE_CLOSE. Before calling MPI_FINALIZE, the user is required to close 7 (via MPI_FILE_CLOSE) all files that were opened with MPI_FILE_OPEN. Note that the 8 communicator comm is unaffected by MPI_FILE_OPEN and continues to be usable in all 9 MPI routines (e.g., MPI_SEND). Furthermore, the use of comm will not interfere with I/O 10 behavior.

11 The format for specifying the file name in the filename argument is implementation 12dependent and must be documented by the implementation.

13 14

15

16

17

18 19

20

21

22

23

24

2526

 31

32 33

3435

36

37 38

39 40

41

4243

44 45

46

4748

An implementation may require that filename include a Advice to implementors. string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End of advice to implementors.)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (End of advice to users.)

Initially, all processes view the file as a linear byte stream, and each process views data 27in its own native representation (no data representation conversion is performed). (POSIX 28files are linear byte streams in the native representation.) The file view can be changed via 29 the MPI_FILE_SET_VIEW routine. 30

The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):

- MPI_MODE_RDONLY read only,
- MPI_MODE_RDWR reading and writing,
- MPI_MODE_WRONLY write only,
- MPI_MODE_CREATE create the file if it does not exist,
- MPI_MODE_EXCL error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE delete file on close,
- MPI_MODE_UNIQUE_OPEN file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL file will only be accessed sequentially,
- MPI_MODE_APPEND set initial position of all file pointers to end of file.

514

Advice to users. C/C++ users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (*End of advice to users.*)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE, and MPI_MODE_EXCL have identical semantics to their POSIX counterparts [38]. Exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, or MPI_MODE_WRONLY, must be specified. It is erroneous to specify MPI_MODE_CREATE or MPI_MODE_EXCL in conjunction with MPI_MODE_RDONLY; it is erroneous to specify MPI_MODE_SEQUENTIAL together with MPI_MODE_RDWR.

The MPI_MODE_DELETE_ON_CLOSE mode causes the file to be deleted (equivalent to performing an MPI_FILE_DELETE) when the file is closed.

The MPI_MODE_UNIQUE_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For MPI_MODE_UNIQUE_OPEN, not opened elsewhere includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI_MODE_UNIQUE_OPEN is specified, the user is responsible for ensuring that no such external events take place. (End of advice to users.)

The MPI_MODE_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI_MODE_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI_FILE_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI_ERR_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 13.2.8, page 521). The constant MPI_INFO_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (End of advice to users.)

Files are opened by default using nonatomic mode file consistency semantics (see Section 13.6.1, page 567). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

Unofficial Draft for Comment Only

 $\overline{7}$

 24

```
1
                 13.2.2 Closing a File
            \mathbf{2}
            3
            4
                 MPI_FILE_CLOSE(fh)
            5
                   INOUT
                             fh
                                                          file handle (handle)
            6
            7
                 int MPI_File_close(MPI_File *fh)
            8
ticket-248T.
                 MPI_File_close(fh, ierror) BIND(C)
            10
                      TYPE(MPI_File), INTENT(INOUT) :: fh
            11
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            12
                 MPI_FILE_CLOSE(FH, IERROR)
            13
                      INTEGER FH, IERROR
            14
            15
                 {void MPI::File::Close() (binding deprecated, see Section 15.2) }
            16
            17
                      MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
                 MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
            18
                 opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
            19
                 MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
            20
            21
                       Advice to users. If the file is deleted on close, and there are other processes currently
            22
                       accessing the file, the status of the file and the behavior of future accesses by these
            23
                       processes are implementation dependent. (End of advice to users.)
            24
            25
                      The user is responsible for ensuring that all outstanding nonblocking requests and
            26
                 split collective operations associated with fh made by a process have completed before that
            27
                 process calls MPI_FILE_CLOSE.
            28
                      The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
            29
                 MPI_FILE_NULL.
            30
            ^{31}
                 13.2.3 Deleting a File
            32
            33
            34
           35
                 MPI_FILE_DELETE(filename, info)
            36
                   IN
                             filename
                                                          name of file to delete (string)
            37
                   IN
                             info
                                                          info object (handle)
            38
            39
  ticket140. 40
                 int MPI_File_delete(const char *filename, MPI_Info info)
ticket-248T. 41
                 MPI_File_delete(filename, info, ierror) BIND(C)
            42
                      CHARACTER(LEN=*), INTENT(IN) :: filename
            43
                      TYPE(MPI_Info), INTENT(IN) :: info
            44
                      INTEGER, OPTIONAL, INTENT(OUT) ::
                                                               ierror
            45
            46
                 MPI_FILE_DELETE(FILENAME, INFO, IERROR)
            47
                      CHARACTER*(*) FILENAME
            48
                      INTEGER INFO, IERROR
```

MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 13.2.8, page 521). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default error handler (see Section 13.7, page 576).

```
13.2.4 Resizing a File
```

```
      MPI_FILE_SET_SIZE(fh, size)

      INOUT
      fh

      file handle (handle)

      IN
      size

      size to truncate or expand file (integer)
```

int MPI_File_set_size(MPI_File fh, MPI_Offset size)

```
MPI_File_set_size(fh, size, ierror) BIND(C)
   TYPE(MPI_File), INTENT(IN) :: fh
   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE

{void MPI::File::Set_size(MPI::Offset size)(binding deprecated, see Section 15.2)
}

MPI_FILE_SET_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI_FILE_SET_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space—use MPI_FILE_PREALLOCATE to force file space to be reserved.

MPI_FILE_SET_SIZE does not affect the individual file pointers or the shared file pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

 $\mathbf{2}$

```
^{23}_{24} ticket-248T.
```

 $47 \\ 48$

1 2 3 4 5 6 7 8 9 10 11 11 12 13	 Advice to users. It is possible for the file pointers to po after a MPI_FILE_SET_SIZE operation truncates a file. The to seeking beyond the current end of file. (<i>End of advice to</i> All nonblocking requests and split collective operations on fh calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE as consistency semantics are concerned, MPI_FILE_SET_SIZE conflicts with operations that access bytes at displacements bet sizes (see Section 13.6.1, page 567). 13.2.5 Preallocating Space for a File 	is is valid, and equivalent o users.) must be completed before SIZE is erroneous. As far is a write operation that
14 15	MPI_FILE_PREALLOCATE(fh, size)	
16	INOUT fh file handle (handle)	
17	IN size size to preallocate file (in	teger)
18 19	<pre>int MPI_File_preallocate(MPI_File fh, MPI_Offset size)</pre>	
ticket-248T. 20 21	<pre>MPI_File_preallocate(fh, size, ierror) BIND(C)</pre>	
22	TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size	
23 24	INTEGER(KIND=MPI_OFFSEI_KIND), INTENI(IN) :: Size INTEGER, OPTIONAL, INTENT(OUT) :: ierror	;
25 26 27 28	MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	dommonated and
29 30	<pre>{void MPI::File::Preallocate(MPI::Offset size)(binding of Section 15.2) }</pre>	ueprecaiea, see
31 32 33 34 35 36 37 38 39 40 41 41 42 43	MPI_FILE_PREALLOCATE ensures that storage space is alloc of the file associated with fh. MPI_FILE_PREALLOCATE is colled group must pass identical values for size. Regions of the file t written are unaffected. For newly allocated regions of the file, N has the same effect as writing undefined data. If size is larger that file size increases to size. If size is less than or equal to the curre unchanged. The treatment of file pointers, pending nonblocking accesses, same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL the file was opened, it is erroneous to call this routine. Advice to users. In some implementations, file preallocation	ective; all processes in the that have previously been MPI_FILE_PREALLOCATE on the current file size, the ent file size, the file size is and file consistency is the mode was specified when
43 44 45 46 47 48	of advice to users.)	

13.2.6 Querying the Size of a File 1 $\mathbf{2}$ 4 MPI_FILE_GET_SIZE(fh, size) 5 IN fh file handle (handle) 6 OUT size size of the file in bytes (integer) 7 8 9 int MPI_File_get_size(MPI_File fh, MPI_Offset *size) 10 ticket-248T. MPI_File_get_size(fh, size, ierror) BIND(C) 11 TYPE(MPI_File), INTENT(IN) :: fh 12INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI_FILE_GET_SIZE(FH, SIZE, IERROR) 16INTEGER FH, IERROR 17INTEGER(KIND=MPI_OFFSET_KIND) SIZE 18 {MPI::Offset MPI::File::Get_size() const(binding deprecated, see Section 15.2) } 1920MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with 21the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a 22 data access operation (see Section 13.6.1, page 567). 23 24 13.2.7 Querying File Parameters 252627MPI_FILE_GET_GROUP(fh, group) 28 IN fh file handle (handle) 29 30 OUT group group which opened the file (handle) 31 32 int MPI_File_get_group(MPI_File fh, MPI_Group *group) 33ticket-248T. 34 MPI_File_get_group(fh, group, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh 35TYPE(MPI_Group), INTENT(OUT) :: group 36 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI_FILE_GET_GROUP(FH, GROUP, IERROR) 39 INTEGER FH, GROUP, IERROR 40 41 {MPI::Group MPI::File::Get_group() const(binding deprecated, see Section 15.2) } 42MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to 43 open the file associated with fh. The group is returned in group. The user is responsible for 44 freeing group. 4546

```
1
                 MPI_FILE_GET_AMODE(fh, amode)
           2
                  IN
                            fh
                                                       file handle (handle)
            3
                  OUT
                            amode
                                                       file access mode used to open the file (integer)
            4
           5
            6
                 int MPI_File_get_amode(MPI_File fh, int *amode)
ticket-248T. 7
                MPI_File_get_amode(fh, amode, ierror) BIND(C)
            8
                     TYPE(MPI_File), INTENT(IN) ::
                                                       fh
           9
                     INTEGER, INTENT(OUT) :: amode
           10
                     INTEGER, OPTIONAL, INTENT(OUT) ::
                                                            ierror
           11
           12
                MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
                     INTEGER FH, AMODE, IERROR
           13
           14
                 {int MPI::File::Get_amode() const(binding deprecated, see Section 15.2) }
           15
           16
                     MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with
           17
                 fh.
           18
                 Example 13.1 In Fortran 77, decoding an amode bit vector will require a routine such as
           19
                 the following:
           20
           21
                       SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
           22
                 !
           23
                     TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
                 T
           24
                     IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
                 !
           25
                 !
           26
                       INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
           27
                       BIT_FOUND = 0
           28
                       CP_AMODE = AMODE
           29
                  100 CONTINUE
           30
                       LBIT = 0
           31
                       HIFOUND = 0
           32
                       DO 20 L = MAX_BIT, 0, -1
           33
                          MATCHER = 2**L
           34
                           IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
           35
                              HIFOUND = 1
           36
                              LBIT = MATCHER
           37
                              CP_AMODE = CP_AMODE - MATCHER
           38
                           END IF
           39
                   20 CONTINUE
           40
                       IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
           41
                       IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
           42
                            CP_AMODE .GT. 0) GO TO 100
           43
                       END
           44
           45
                     This routine could be called successively to decode amode, one bit at a time. For
           46
                 example, the following code fragment would check for MPI_MODE_RDONLY.
           47
```

520

CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)			
IF (BIT_FOUND .EQ. 1) THEN			
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE			
ELSE			
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE			
END IF			

13.2.8 File Info

Hints specified via info (see Section 9, page 383) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. However, hints do not change the semantics of any of the I/O interfaces. In other words, an implementation is free to ignore all hints. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

MPI_FILE_SET_INFO(fh, info)

INOUT	fh	file handle (handle)
IN	info	info object (handle)

int MPI_File_set_info(MPI_File fh, MPI_Info info)
MPI_File_set_info(fh, info, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(MPI_Info), INTENT(IN) :: info
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR

{void MPI::File::Set_info(const MPI::Info& info)(binding deprecated, see Section 15.2) }

MPI_FILE_SET_INFO sets new values for the hints of the file associated with fh. MPI_FILE_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Unofficial Draft for Comment Only

 $_{34}$ ticket-248T.

1 Advice to users. Many info items that an implementation can use when it creates or $\mathbf{2}$ opens a file cannot easily be changed once the file has been created or opened. Thus, 3 an implementation may ignore hints issued in this call that it would have accepted in 4 an open call. (End of advice to users.) 56 $\overline{7}$ MPI_FILE_GET_INFO(fh, info_used) 8 9 IN fh file handle (handle) 10 OUT info_used new info object (handle) 11 12int MPI_File_get_info(MPI_File fh, MPI_Info *info_used) 13ticket-248T. 14MPI_File_get_info(fh, info_used, ierror) BIND(C) 15TYPE(MPI_File), INTENT(IN) :: fh 16TYPE(MPI_Info), INTENT(OUT) :: info_used 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI_FILE_GET_INFO(FH, INFO_USED, IERROR) 19INTEGER FH, INFO_USED, IERROR 2021{MPI:::Info MPI::File::Get_info() const(binding deprecated, see Section 15.2) } 22MPI_FILE_GET_INFO returns a new info object containing the hints of the file associ-23ated with fh. The current setting of all hints actually used by the system related to this open 24 file is returned in info_used. If no such hints exist, a handle to a newly created info object 25is returned that contains no key/value pair. The user is responsible for freeing info_used 26via MPI_INFO_FREE. 2728 Advice to users. The info object returned in info_used will contain all hints currently 29active for this file. This set of hints may be greater or smaller than the set of hints 30 passed in to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, as 31the system may not recognize some hints set by the user, and may recognize other 32 hints that the user has not set. (End of advice to users.) 33 34Reserved File Hints 35 36 Some potentially useful hints (info key values) are outlined below. The following key values 37 are reserved. An implementation is not required to interpret these key values, but if it does 38interpret the key value, it must provide the functionality described. (For more details on 39 "info," see Section 9, page 383.) 40These hints mainly affect access patterns and the layout of data on parallel I/O devices. 41 For each hint name introduced, we describe the purpose of the hint, and the type of the hint 42value. The "[SAME]" annotation specifies that the hint values provided by all participating 43processes must be identical; otherwise the program is erroneous. In addition, some hints are 44context dependent, and are only used by an implementation at specific times (e.g., file_perm 45is only useful during file creation). 4647access_style (comma separated list of strings): This hint specifies the manner in which 48 the file will be accessed until the file is closed or until the access_style key value is altered. The hint value is a comma separated list of the following: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random.

- collective_buffering (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are true and false. Collective buffering parameters are further directed via additional hints: cb_block_size, cb_buffer_size, and cb_nodes.
- cb_block_size (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (CYCLIC) pattern.
- cb_buffer_size (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of cb_block_size.
- cb_nodes (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- chunked (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- chunked_item (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- chunked_size (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- filename (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.
- file_perm (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode that includes MPI_MODE_CREATE. The set of valid values for this key is implementation dependent.
- io_node_list (comma separated list of strings) [SAME]: This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.
- nb_proc (integer) [SAME]: This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10 11

12

13

14

15

16 17

18

19

20

21

22

23

 24

25 26

27

28

29

30

31 32

33

34

35

36

37

38

39

40

41 42

43

44

 $45 \\ 46$

47

	$\frac{1}{2}$		· · · ·	AME]: This hint specifies the number of I/O devices in the nost relevant when the file is created.
	3 4 5		τ ο / ι	AME]: This hint specifies the number of I/O devices that the across, and is relevant only when the file is created.
	6 7 8 9 10	for dev	this file. The strip vice before progress	ME]: This hint specifies the suggested striping unit to be used ing unit is the amount of consecutive data assigned to one I/O ing to the next device, when striping across a number of devices. es. This hint is relevant only when the file is created.
	11 12 13 14	13.3 I	File Views	
	15		E SET VIEW/(fb d	isp, etype, filetype, datarep, info)
	16	INOUT		
	17 18			file handle (handle)
	19	IN	disp	displacement (integer)
	20	IN	etype	elementary datatype (handle)
	21	IN	filetype	filetype (handle)
	22 23	IN	datarep	data representation (string)
	24	IN	info	info object (handle)
ticket140	25 26 • 27	int MPI		PI_File fh, MPI_Offset disp, MPI_Datatype etype, /pe filetype, <mark>const</mark> char *datarep, MPI_Info info)
ticket229.2 ticket-248T	• 28 • 29 30 31 32 33	TYPI INTI TYPI CHAI	E(MPI_File), INT EGER(KIND=MPI_OF E(MPI_Datatype), RACTER(LEN=*), I	<pre>lisp, etype, filetype, datarep, info, ierror) BIND(C) ENT(IN) :: fh FSET_KIND), INTENT(IN) :: disp INTENT(IN) :: etype, filetype NTENT(IN) :: datarep ENT(IN) :: info</pre>
	34			INTENT(OUT) :: ierror
	35 36 37 38 39	INT CHAI		
	40 41 42 43	{void MI	const MPI	<pre>ew(MPI::Offset disp, const MPI::Datatype& etype, :Datatype& filetype, const char* datarep, ::Info& info)(binding deprecated, see Section 15.2) }</pre>
	43 44 45 46	The start to proces	t of the view is set sses is set to filety	IEW routine changes the process's view of the data in the file. to disp; the type of data is set to etype; the distribution of data be; and the representation of data in the file is set to datarep.
	47 48		,	Γ_VIEW resets the individual file pointers and the shared file SET_VIEW is collective; the values for datarep and the extents

of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4, page 11), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 13.5.1, page 558 for further details.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

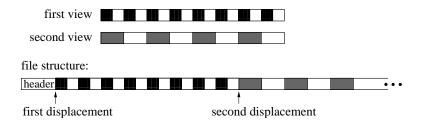


Figure 13.3: Displacements

(End of advice to users.)

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of **etypes**; file pointers point to the beginning of etypes.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 24

 $41 \\ 42$

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (see Section 13.5, page 556). (End of advice to users.)

A filetype is either a single etype or a derived MPI datatype constructed from multiple 5instances of the same etype. In addition, the extent of any hole in the filetype must be 6 a multiple of the etype's extent. These displacements are not required to be distinct, but $\overline{7}$ they cannot be negative, and they must be monotonically nondecreasing. 8

If the file is opened for writing, neither the etype nor the filetype is permitted to contain 9 10 overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different 11 processes may still overlap each other. 12

If filetype has holes in it, then the data in the holes is inaccessible to the calling process. 13 However, the disp, etype and filetype arguments can be changed via future calls to 14MPI_FILE_SET_VIEW to access a different part of the file. 15

16It is erroneous to use absolute addresses in the construction of the etype and filetype.

17The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8, page 521). The constant 18 MPI_INFO_NULL refers to the null info and can be used when no info needs to be specified. 19The datarep argument is a string that specifies the representation of data in the file. 2021See the file interoperability section (Section 13.5, page 556) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective 23operations on fh have been completed before calling MPI_FILE_SET_VIEW—otherwise, the 24 call to MPI_FILE_SET_VIEW is erroneous. 25

int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,

MPI_Datatype *filetype, char *datarep)

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

```
IN
          fh
                                           file handle (handle)
OUT
           disp
                                           displacement (integer)
OUT
          etype
                                           elementary datatype (handle)
OUT
          filetype
                                           filetype (handle)
OUT
          datarep
                                           data representation (string)
```

38 ticket229.2.

ticket-248T. 39

```
MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) BIND(C)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
42
         TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
43
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
46
         INTEGER FH, ETYPE, FILETYPE, IERROR
47
         CHARACTER*(*) DATAREP
```

1

2

3

4

22

2627

28

29 30

 31

32

33

34

35 36

37

INTEGER(KIND=MPI_OFFSET_KIND) DISP

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

13.4 Data Access

13.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 13.1.

positioning	synchronism	со	ordination	27
		noncollective	collective	28
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL	29
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL	30
	nonblocking &	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN	31
	split collective		MPI_FILE_READ_AT_ALL_END	32
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN	
			MPI_FILE_WRITE_AT_ALL_END	33
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL	34
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL	35
	nonblocking \mathfrak{E}	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN	36
	split collective		MPI_FILE_READ_ALL_END	37
		MPI_FILE_IWRITE	MPI_FILE_WRITE_ALL_BEGIN	38
			MPI_FILE_WRITE_ALL_END	39
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED	40
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED	
	nonblocking \mathfrak{E}	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN	
	split collective		MPI_FILE_READ_ORDERED_END	42
		MPI_FILE_IWRITE_SHARED	MPI_FILE_WRITE_ORDERED_BEGI	
			MPI_FILE_WRITE_ORDERED_END	44

Table 13.1: Data access routines

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1 use individual file pointers. The MPI equivalents are MPI_FILE_READ and $\mathbf{2}$ MPI_FILE_WRITE. 3 Implementations of data access routines may buffer data to improve performance. This 4 does not affect reads, as the data is always available in the user's buffer after a read operation $\mathbf{5}$ completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee 6 that data has been transferred to the storage device. $\overline{7}$ 8 Positioning 9 MPI provides three types of positioning for data access routines: explicit offsets, individual 10 file pointers, and shared file pointers. The different positioning methods may be mixed 11 within the same program and do not affect each other. 12The data access routines that accept explicit offsets contain _AT in their name (e.g., 13 MPI_FILE_WRITE_AT). Explicit offset operations perform data access at the file position 14given directly as an argument—no file pointer is used nor updated. Note that this is not 15equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. 16Operations with explicit offsets are described in Section 13.4.2, page 530. 17The names of the individual file pointer routines contain no positional qualifier (e.g., 18 MPI_FILE_WRITE). Operations with individual file pointers are described in Section 13.4.3, 19page 534. The data access routines that use shared file pointers contain _SHARED or 20_ORDERED in their name (e.g., MPI_FILE_WRITE_SHARED). Operations with shared file 21pointers are described in Section 13.4.4, page 542. 22The main semantic issues with MPI-maintained file pointers are how and when they are 23updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to 24the next data item after the last one that is accessed by the operation. In a nonblocking or 25split collective operation, the pointer is updated by the call that initiates the I/O, possibly 26before the access completes. 27

More formally,

30 31 32

33

34

35

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of predefined datatypes in the typemap of X, and *old_file_offset* is the value of the implicit offset before the call. The file position, new_file_offset , is in terms of a count of etypes relative to the current view.

36 37 Synchronism

MPI supports blocking and nonblocking I/O routines.

A $blocking \, \mathrm{I/O}$ call will not return until the $\mathrm{I/O}$ request is completed.

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$

A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete.
 Given suitable hardware, this allows the transfer of data out/in the user's buffer to proceed
 concurrently with computation. A separate *request complete* call (MPI_WAIT, MPI_TEST,
 or any of their variants) is needed to complete the I/O request, i.e., to confirm that the data
 has been read or written and that it is safe for the user to reuse the buffer. The nonblocking
 versions of the routines are named MPI_FILE_IXXX, where the I stands for immediate.

It is erroneous to access the local buffer of a nonblocking data access operation, or to
 use that buffer as the source or target of other communications, between the initiation and
 completion of the operation.

³⁸ 39

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 13.4.5, page 548).

Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4, page 570, for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 on page 29 and Section 4.1.11 on page 119. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 524). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in — [subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.16, pages 673 and 679. subsections "Problems Due to]Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 673-676 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 16.2.16 to 16.2.19 on pages 679 to 688 about 40 41 42 ticket238-J. 43 44 ⁴⁵ ticket238-J. ⁴⁶ ticket236-H. ⁴⁷ ticket238-J. 48

Unofficial Draft for Comment Only

1 2

3 4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

17 18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

	1			s", "Code Movements and Register Optimization", "Temporary
	2	Data	Movements" and	"Permanent Data Movements". (End of advice to users.)
	3 4	For b	locking routines	status is returned directly. For nonblocking routines and split
	5		0 ,	s returned when the operation is completed. The number of
	6		/	ned elements accessed by the calling process can be extracted
	7		-	ET_COUNT and MPI_GET_ELEMENTS, respectively. The inter-
	8			field is the same as for other operations — normally undefined,
	9	but meani	ngful if an MPI r	outine returns $MPI_ERR_IN_STATUS$. The user can pass (in C
	10	and Fortra	an) MPI_STATUS_I	GNORE in the status argument if the return value of this argu-
	11			+, the status argument is optional. The status can be passed
	12			to determine if the operation was cancelled. All other fields of
	13	status are		
	14			um can detect the end of file by noting that the amount of data
	15			t requested. Writing past the end of file increases the file size.
	16 17			d will be the amount requested, unless an error is raised (or a
	18	read reach	es the end of file)	
	19	13.4.2 D	ata Access with I	Evaluate Officeta
	20	13.4.2 D		
	21			mode was specified when the file was opened, it is erroneous to
	22	call the ro	utines in this sect	ion.
	23			
	24 25	MPI_FILE	_READ_AT(fh, off	set, buf, count, datatype, status)
	25 26	IN	fh	file handle (handle)
	27	IN	offset	file offset (integer)
	28 29	OUT	buf	initial address of buffer (choice)
	29 30	IN	count	number of elements in buffer (integer)
	31	IN	datatype	datatype of each buffer element (handle)
	32	OUT	status	status object (Status)
	33			3 ()
	34	int MPI H	File read at(MP)	I_File fh, MPI_Offset offset, void *buf, int count,
	35	_		pe datatype, MPI_Status *status)
ticket-248T	36 37	NDT THI	·	· · · ·
	38		_read_at(in, of: (MPI_File), INT	<pre>fset, buf, count, datatype, status, ierror) BIND(C) ENT(IN) fb</pre>
	39			FSET_KIND), INTENT(IN) :: offset
	40		(*), DIMENSION(
	41		GER, INTENT(IN)	
	42			INTENT(IN) :: datatype
	43	TYPE	(MPI_Status) ::	status
	44	INTEC	ER, OPTIONAL,	INTENT(OUT) :: ierror
	45	MDI EILE	READ AT(EH OF	FSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
	46		BUF(*)	, Sor, Cooki, BRIRITE, BIRIOD, ILHUUR/
	47 48	• -		DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
	48			, ,

INTEG	1		
{void MPI	2		
(3		
	5		
∫void MPT	··File··Read at (MPT··Off	set offset, void* buf, int count,	6
l nora un r		atatype) (binding deprecated, see Section 15.2)	7
	}		8
		eginning at the position specified by offset.	9
	TEL_READ_AT reads a life be	eginning at the position specified by onset.	10
			11 12
MPI_FILE_	READ_AT_ALL(fh, offset, buf	f, count, datatype, status)	12
IN	fh	file handle (handle)	14
IN	offset	file offset (integer)	15
OUT	buf	initial address of buffer (choice)	16
			17
IN	count	number of elements in buffer (integer)	18
IN	datatype	datatype of each buffer element (handle)	19 20
OUT	status	status object (Status)	20
			22
int MPI_F		fh, MPI_Offset offset, void *buf,	23
	int count, MP1_Datat	ype datatype, MPI_Status *status)	24 ticket-248T.
MPI_File_	read_at_all(fh, offset, 1	buf, count, datatype, status, ierror)	25
	BIND(C)		26
	<pre>MPI_File), INTENT(IN) ::</pre>		27
	ER(KIND=MPI_OFFSET_KIND)		28 29
	*), DIMENSION() :: b ER, INTENT(IN) :: count	ui	30
	MPI_Datatype), INTENT(IN)):: datatype	31
	MPI_Status) :: status		32
	ER, OPTIONAL, INTENT(OUT)) :: ierror	33
MPT FILE	READ AT ALL (FH OFFSET 1	BUF, COUNT, DATATYPE, STATUS, IERROR)	34
	> BUF(*)		35
• 1		STATUS(MPI_STATUS_SIZE), IERROR	36 37
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	38
{void MPT	::File::Read at all(MPT:	:Offset offset, void* buf, int count,	39
(datatype, MPI::Status& status)(binding	40
	deprecated, see Section	• •	41
Juoid MDT	··File··Read at all(MPT.	:Offset offset, void* buf, int count,	42
l vora ur r		(binding deprecated, see Section 15.2)	43
	}	J1 · (······· J ··· J ···· J ····· J ····· J ········ J ··········	44 45
		ective version of the blocking MPI_FILE_READ_AT	46
interface.	ILL_ILAU_AI_ALL IS a COILE	conversion of the Diocking WIFI_FILE_READ_AI	47
Interface.			

```
1
                 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
            \mathbf{2}
                   INOUT
                             fh
                                                         file handle (handle)
            3
                             offset
                   IN
                                                         file offset (integer)
            4
            5
                             buf
                                                         initial address of buffer (choice)
                   IN
            6
                   IN
                             count
                                                         number of elements in buffer (integer)
            7
                   IN
                             datatype
                                                         datatype of each buffer element (handle)
            8
            9
                   OUT
                                                         status object (Status)
                             status
            10
           11
  ticket140.
                 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
            12
                                 int count, MPI_Datatype datatype, MPI_Status *status)
ticket-248T. 13
                 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
            14
                      TYPE(MPI_File), INTENT(IN) :: fh
           15
                      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
            16
                      TYPE(*), DIMENSION(...), INTENT(IN) :: buf
            17
                      INTEGER, INTENT(IN) :: count
            18
                      TYPE(MPI_Datatype), INTENT(IN) :: datatype
            19
                      TYPE(MPI_Status) :: status
           20
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           21
           22
                 MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
           23
                      <type> BUF(*)
           24
                      INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
           25
                      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
           26
                 {void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
           27
                                 const MPI::Datatype& datatype, MPI::Status& status) (binding
           28
                                deprecated, see Section 15.2 }
           29
           30
                 {void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
           ^{31}
                                const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
            32
                                 ł
           33
           34
                      MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.
           35
           36
                 MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)
           37
           38
                   INOUT
                             fh
                                                         file handle (handle)
           39
                   IN
                             offset
                                                         file offset (integer)
           40
                             buf
                   IN
                                                         initial address of buffer (choice)
           41
           42
                   IN
                             count
                                                         number of elements in buffer (integer)
           43
                   IN
                             datatype
                                                         datatype of each buffer element (handle)
           44
                   OUT
                             status
                                                         status object (Status)
           45
            46
                 int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
  ticket140. 47
                                 int count, MPI_Datatype datatype, MPI_Status *status)
            48
```

ticket-

ticket 229.1.				1		
ticket-248T.	MPI_File		offset, buf, count, datatype, status, ierror)	2		
	זרעייי	BIND(C)	(TN) fb	3		
		E(MPI_File), INTENT	T_KIND), INTENT(IN) :: offset	4 5		
			, INTENT(IN) :: buf	6		
	INTEGER, INTENT(IN) :: count					
	TYPE	8				
	TYPE	E(MPI_Status) :: s	tatus	9		
	INTE	EGER, OPTIONAL, INT	ENT(OUT) :: ierror	10		
	MPT FTLF	E WRITE AT ALL(FH.	OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	11		
		<pre>De> BUF(*)</pre>		12		
	• -		ATYPE, STATUS(MPI_STATUS_SIZE), IERROR	13		
	INTE	EGER(KIND=MPI_OFFSE	T_KIND) OFFSET	14 15		
	{void ME	DI.File.Write at	all(MPI::Offset offset, const void* buf,	16		
	(VOIG III		onst MPI::Datatype& datatype,	17		
			status) (binding deprecated, see Section 15.2) }	18		
	(·) M			19		
	{void MF		all(MPI::Offset offset, const void* buf, onst MPI::Datatype& datatype)(binding deprecated, see	20		
		mst MF1Datatype& datatype)(omunig deprecated, see	21			
		22				
	MPI.	23				
	MPI_FILE	24 25				
		26				
	MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)					
	IN	fh	file handle (handle)	28		
	IN	offset	file offset (integer)	29		
	OUT	buf	initial address of buffer (choice)	30		
				31		
	IN	count	number of elements in buffer (integer)	32 33		
	IN	datatype	datatype of each buffer element (handle)	34		
	OUT	request	request object (handle)	35		
				36		
	int MPI_	37				
		³⁸ ₃₉ ticket-248T.				
	MPI_File					
		40				
	TYPE	E(MPI_File), INTENT	(IN) :: fh	41 42		
			T_KIND), INTENT(IN) :: offset	43		
	TYPE	44				
	INTE	45				
	TYPE(MPI_Datatype), INTENT(IN) :: datatype					
	TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
	±.,11	, 0.1100000, 101		48		

```
1
                MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
           \mathbf{2}
                     <type> BUF(*)
           3
                     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
            4
                     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
           5
                 {MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count,
           6
                                const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
            7
                                }
            8
           9
                     MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
           10
           11
                 MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
           12
           13
                   INOUT
                            fh
                                                        file handle (handle)
           14
                   IN
                            offset
                                                        file offset (integer)
           15
                            buf
                                                        initial address of buffer (choice)
                   IN
           16
           17
                   IN
                            count
                                                        number of elements in buffer (integer)
           18
                   IN
                            datatype
                                                        datatype of each buffer element (handle)
           19
                   OUT
                            request
                                                        request object (handle)
           20
           21
  ticket140. 22
                 int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
           23
                                int count, MPI_Datatype datatype, MPI_Request *request)
ticket-248T. 24
                 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
           25
                                BIND(C)
           26
                     TYPE(MPI_File), INTENT(IN) :: fh
           27
                     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
           28
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
           29
                     INTEGER, INTENT(IN) :: count
           30
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           31
                     TYPE(MPI_Request), INTENT(OUT) :: request
           32
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           33
           34
                 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
           35
                     <type> BUF(*)
           36
                     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
           37
                     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
           38
                 {MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf,
           39
                                int count, const MPI::Datatype& datatype) (binding deprecated, see
           40
                                Section 15.2 }
           41
           42
                     MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
           43
           44
                 13.4.3 Data Access with Individual File Pointers
           45
                 MPI maintains one individual file pointer per process per file handle. The current value
           46
                 of this pointer implicitly specifies the offset in the data access routines described in this
           47
           48
```

section. These routines only use and update the individual file pointers maintained by MPI. The shared file pointer is not used nor updated.

The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 530, with the following modification:

• the offset is defined to be the current value of the MPI-maintained individual file pointer.

After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.

MPI_FILE_READ(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)	10
OUT	buf	initial address of buffer (choice)	18
IN	count	number of elements in buffer (integer)	19
IN	datatype	datatype of each buffer element (handle)	20
OUT	status	status object (Status)	21 22

MPI_File_read(fh, buf, count, datatype, status, ierror) BIND(C)
 TYPE(MPI_File), INTENT(IN) :: fh
 TYPE(*), DIMENSION(..) :: buf
 INTEGER, INTENT(IN) :: count
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FILE_READ reads a file using the individual file pointer.

Example 13.2 The following Fortran code fragment is an example of reading a file until the end of file is reached:

 24

 $44 \\ 45$

 $^{25}_{26}$ ticket-248T.

```
1
                !
                     Read a preexisting input file until all data has been read.
           \mathbf{2}
                !
                     Call routine "process_input" if all requested data is read.
           3
                1
                     The Fortran 90 "exit" statement exits the loop.
           4
           5
                                  bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
                       integer
           6
                       parameter (bufsize=100)
           7
                       real
                                  localbuffer(bufsize)
           8
                       integer (kind=MPI_OFFSET_KIND) zero
           9
           10
                       zero = 0
           11
           12
                       call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
           13
                                             MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
           14
                       call MPI_FILE_SET_VIEW( myfh, zero, MPI_REAL, MPI_REAL, 'native', &
           15
                                             MPI_INFO_NULL, ierr )
           16
                       totprocessed = 0
           17
                       do
           18
                          call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
           19
                                                status, ierr )
           20
                          call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
           21
                          call process_input( localbuffer, numread )
           22
                          totprocessed = totprocessed + numread
           23
                          if ( numread < bufsize ) exit
           24
                       enddo
           25
           26
                       write(6,1001) numread, bufsize, totprocessed
           27
                1001 format( "No more data: read", I3, "and expected", I3, &
           28
                                "Processed total of", I6, "before terminating job." )
           29
           30
                       call MPI_FILE_CLOSE( myfh, ierr )
           31
           32
           33
                MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
           34
                  INOUT
                           fh
           35
                                                       file handle (handle)
           36
                  OUT
                           buf
                                                      initial address of buffer (choice)
           37
                  IN
                           count
                                                      number of elements in buffer (integer)
           38
           39
                  IN
                                                      datatype of each buffer element (handle)
                           datatype
           40
                  OUT
                           status
                                                      status object (Status)
           41
           42
                int MPI_File_read_all(MPI_File fh, void *buf, int count,
           43
                               MPI_Datatype datatype, MPI_Status *status)
ticket-248T. ^{44}
           45
                MPI_File_read_all(fh, buf, count, datatype, status, ierror) BIND(C)
           46
                     TYPE(MPI_File), INTENT(IN) :: fh
           47
                     TYPE(*), DIMENSION(..) :: buf
           48
                     INTEGER, INTENT(IN) :: count
```

		NTENT(IN) :: datatype	1
	E(MPI_Status) ::		2
INTE	EGER, OPTIONAL, IN	TENT(OUT) :: ierror	3
MPT FTLF	READ ALL(FH. BUF	, COUNT, DATATYPE, STATUS, IERROR)	4
	<pre>be> BUF(*)</pre>	,,,,	5
• -		TATYPE, STATUS(MPI_STATUS_SIZE), IERROR	6 7
			8
{void MH		(void* buf, int count,	9
		Datatype& datatype, MPI::Status& status)(binding	10
	aeprecatea, se	$e Section (15.2) \}$	11
{void MF	PI::File::Read_all	(void* buf, int count,	12
	const MPI::I	Datatype& datatype) (binding deprecated, see Section 15.2)	13
	}		14
MPI	FILE READ ALL is	a collective version of the blocking MPI_FILE_READ interface.	15
			16
			17
MPI_FILE	E_WRITE(fh, buf, cou	ınt, datatype, status)	18
INOUT	fh	file handle (handle)	19
IN	buf	initial address of buffer (choice)	20
			21
IN	count	number of elements in buffer (integer)	22
IN	datatype	datatype of each buffer element (handle)	23 24
OUT	status	status object (Status)	25
			26
int MPI_	_File_write(MPI_Fi	le fh, const void *buf, int count,	$_{27}$ ticket 140.
	MPI_Datatype	e datatype, MPI_Status *status)	²⁸
MDT File	write(fh buf c	count, datatype, status, ierror) BIND(C)	$_{29}^{20}$ ticket-248T.
	E(MPI_File), INTEN		30
), INTENT(IN) :: buf	31
	EGER, INTENT(IN) :		32
		NTENT(IN) :: datatype	33
TYPE	E(MPI_Status) ::	status	34
INTE	EGER, OPTIONAL, IN	TENT(OUT) :: ierror	35
	י שמדייב בי מוב מ	COUNT, DATATYPE, STATUS, IERROR)	36
	<pre>be> BUF(*)</pre>	JONI, DATATIFE, STATOS, TEMION/	37 38
01		TATYPE, STATUS(MPI_STATUS_SIZE), IERROR	39
			40
{void MF		nst void* buf, int count,	41
		Datatype& datatype, MPI::Status& status)(binding	42
	aeprecatea, se	$e Section (15.2) \}$	43
{void MF	PI::File::Write(co	nst void* buf, int count,	44
	const MPI::I	Datatype& datatype) (binding deprecated, see Section 15.2)	45
	}		46
MPI	_FILE_WRITE writes	a file using the individual file pointer.	47
			48

```
1
                 MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
            \mathbf{2}
                   INOUT
                             fh
                                                         file handle (handle)
            3
                   IN
                             buf
                                                         initial address of buffer (choice)
            4
            5
                   IN
                             count
                                                         number of elements in buffer (integer)
            6
                   IN
                                                         datatype of each buffer element (handle)
                             datatype
            7
                   OUT
                             status
                                                         status object (Status)
            8
            9
  ticket140. 10
                 int MPI_File_write_all(MPI_File fh, const void *buf, int count,
            11
                                MPI_Datatype datatype, MPI_Status *status)
ticket-248T. 12
                 MPI_File_write_all(fh, buf, count, datatype, status, ierror) BIND(C)
            13
                     TYPE(MPI_File), INTENT(IN) :: fh
           14
                     TYPE(*), DIMENSION(...), INTENT(IN) :: buf
           15
                     INTEGER, INTENT(IN) :: count
            16
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
            17
                     TYPE(MPI_Status) :: status
            18
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            19
           20
                 MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
           21
                      <type> BUF(*)
           22
                     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
           23
                 {void MPI::File::Write_all(const void* buf, int count,
           24
                                const MPI::Datatype& datatype, MPI::Status& status) (binding
           25
                                deprecated, see Section 15.2 }
            26
           27
                 {void MPI::File::Write_all(const void* buf, int count,
           28
                                const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
           29
                                ł
           30
                     MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
           ^{31}
                 face.
           32
           33
           34
                 MPI_FILE_IREAD(fh, buf, count, datatype, request)
           35
                   INOUT
                             fh
                                                         file handle (handle)
           36
           37
                   OUT
                             buf
                                                         initial address of buffer (choice)
           38
                   IN
                             count
                                                         number of elements in buffer (integer)
           39
                   IN
                             datatype
                                                         datatype of each buffer element (handle)
           40
           41
                   OUT
                                                         request object (handle)
                             request
           42
           43
                 int MPI_File_iread(MPI_File fh, void *buf, int count,
           44
                                MPI_Datatype datatype, MPI_Request *request)
ticket-248T. 45
                 MPI_File_iread(fh, buf, count, datatype, request, ierror) BIND(C)
            46
                     TYPE(MPI_File), INTENT(IN) :: fh
            47
                     TYPE(*), DIMENSION(...), ASYNCHRONOUS ::
                                                                    buf
            48
```

```
1
    INTEGER, INTENT(IN) :: count
                                                                                      \mathbf{2}
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                      3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      4
                                                                                      5
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                      6
    <type> BUF(*)
                                                                                      7
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                      8
                                                                                      9
{MPI::Request MPI::File::Iread(void* buf, int count,
                                                                                      10
              const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
                                                                                      11
              }
                                                                                      12
    MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
                                                                                      13
                                                                                      14
Example 13.3 The following Fortran code fragment illustrates file pointer update seman-
                                                                                      15
tics:
                                                                                      16
                                                                                      17
!
    Read the first twenty real words in a file into two local
                                                                                      18
!
    buffers. Note that when the first MPI_FILE_IREAD returns,
                                                                                      19
!
    the file pointer has been updated to point to the
                                                                                      20
I.
    eleventh real word in the file.
                                                                                      21
                                                                                      22
                 bufsize, req1, req2
      integer
                                                                                      23
      integer, dimension(MPI_STATUS_SIZE) :: status1, status2
                                                                                      24
      parameter (bufsize=10)
                                                                                      25
                 buf1(bufsize), buf2(bufsize)
      real
                                                                                      26
      integer (kind=MPI_OFFSET_KIND) zero
                                                                                      27
                                                                                      28
      zero = 0
                                                                                      29
      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                                                                                      30
                            MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
                                                                                      31
      call MPI_FILE_SET_VIEW( myfh, zero, MPI_REAL, MPI_REAL, 'native', &
                                                                                      32
                           MPI_INFO_NULL, ierr )
                                                                                      33
      call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
                                                                                      34
                             req1, ierr )
                                                                                      35
      call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
                                                                                      36
                             req2, ierr )
                                                                                      37
                                                                                      38
      call MPI_WAIT( req1, status1, ierr )
                                                                                      39
      call MPI_WAIT( req2, status2, ierr )
                                                                                      40
                                                                                      41
      call MPI_FILE_CLOSE( myfh, ierr )
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
                                                                                      48
```

```
1
                 MPI_FILE_IWRITE(fh, buf, count, datatype, request)
           \mathbf{2}
                   INOUT
                            fh
                                                        file handle (handle)
            3
                   IN
                            buf
                                                        initial address of buffer (choice)
            4
           5
                   IN
                            count
                                                        number of elements in buffer (integer)
            6
                   IN
                                                        datatype of each buffer element (handle)
                            datatype
            7
                   OUT
                            request
                                                        request object (handle)
            8
           9
  ticket140. 10
                 int MPI_File_iwrite(MPI_File fh, const void *buf, int count,
           11
                                MPI_Datatype datatype, MPI_Request *request)
ticket-248T. 12
                 MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)
           13
                     TYPE(MPI_File), INTENT(IN) :: fh
           14
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
           15
                     INTEGER, INTENT(IN) :: count
           16
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           17
                     TYPE(MPI_Request), INTENT(OUT) ::
                                                            request
           18
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           19
           20
                 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
           21
                     <type> BUF(*)
           22
                     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
           23
                 {MPI::Request MPI::File::Iwrite(const void* buf, int count,
           24
                                const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
           25
                                }
           26
           27
                     MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
           28
           29
                 MPI_FILE_SEEK(fh, offset, whence)
           30
           ^{31}
                   INOUT
                            fh
                                                        file handle (handle)
           32
                   IN
                            offset
                                                        file offset (integer)
           33
                   IN
                            whence
                                                        update mode (state)
           34
           35
           36
                 int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
ticket-248T. 37
                 MPI_File_seek(fh, offset, whence, ierror) BIND(C)
           38
                     TYPE(MPI_File), INTENT(IN) :: fh
           39
                     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
           40
                     INTEGER, INTENT(IN) :: whence
           41
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           42
           43
                 MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
           44
                     INTEGER FH, WHENCE, IERROR
           45
                     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
           46
                 {void MPI::File::Seek(MPI::Offset offset, int whence) (binding deprecated, see
           47
                                Section 15.2 }
           48
```

MPI following	1 2				
• MP	$\frac{3}{4}$				
• MP	5				
• MP	6 7				
	• MPI_SEEK_END: the pointer is set to the end of file plus offset				
The	9				
a negative position in the view.					
MPI_FIL	12				
			13		
IN	fh	file handle (handle)	14		
OUT	offset	offset of individual pointer (integer)	15		
		16 17			
<pre>int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)</pre>					
MPT File	<pre>/PI_File_get_position(fh, offset, ierror) BIND(C)</pre>				
			19 20		
TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset					
		, INTENT(OUT) :: ierror	21 22		
	23				
MPI_FIL	24				
INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET					
TINT	OFFSEI_KIND) OFFSEI	26			
{MPI::0:	ffset MPI::File	e::Get_position() const(binding deprecated, see Section 15.2)	27		
	}		28		
MPI	FILE GET POS	ITION returns, in offset, the current position of the individual file	29		
		ative to the current view.	30		
pointer	n etype units rele		31		
Ad	vice to users.	The offset can be used in a future call to MPI_FILE_SEEK using	32		
wh	$ence = MPI_SEEK$	_SET to return to the current position. To set the displacement to	33		
the	current file point	er position, first convert offset into an absolute byte position using	34		
MF	PI_FILE_GET_BY	TE_OFFSET, then call MPI_FILE_SET_VIEW with the resulting	35		
dis	placement. $(End$	of advice to users.)	36		
			37		
			38 39		
MPI FII	40				
		FFSET(fh, offset, disp)	41		
IN	fh	file handle (handle)	42		
IN	offset	offset (integer)	43		
OUT	disp	absolute byte position of offset (integer)	44		
	·		45		
<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp)</pre>					

1	MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C)
2	TYPE(MPI_File), INTENT(IN) :: fh
3	<pre>INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset</pre>
4	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6	···· · ··· · ··· · ··· · ··· · · · · ·
7	MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
8	INTEGER FH, IERROR
9	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
10	{MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp)
11	const (binding deprecated, see Section 15.2) }
12	const (timuling appreciated, see beenton 10.2)
13	MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte
14	position. The absolute byte position (from the beginning of the file) of offset relative to the
15	current view of fh is returned in disp.
16	
10	13.4.4 Data Access with Shared File Pointers
18	
	MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among
19	processes in the communicator group). The current value of this pointer implicitly specifies
20	the offset in the data access routines described in this section. These routines only use and
21	update the shared file pointer maintained by MPI. The individual file pointers are not used
22	nor updated.
23	The shared file pointer routines have the same semantics as the data access with explicit
24	offset routines described in Section 13.4.2, page 530, with the following modifications:
25	onset is a line accorded in Section 10112, page 556, while the following include to the
26	\bullet the offset is defined to be the current value of the MPI-maintained shared file pointer,
27	
28	• the effect of multiple calls to shared file pointer routines is defined to behave as if the
29	calls were serialized, and
30	• the use of shared file pointer routines is erroneous unless all processes use the same
31	file view.
32	
33	For the noncollective shared file pointer routines, the serialization ordering is not determin-
34	istic. The user needs to use other synchronization means to enforce a specific order.
35	After a shared file pointer operation is initiated, the shared file pointer is updated to
36	point to the next etype after the last one that will be accessed. The file pointer is updated
37	relative to the current view of the file.
38	
39	
40	
41	
42	
43	
43	
45	
46	
40	
48	
40	

10.1. 2.11	11 110 02.55		010			
Noncollecti	1					
			2			
			3			
MPI_FILE_	4					
INOUT	fh	file handle (handle)	5			
	buf		6 7			
OUT		initial address of buffer (choice)	8			
IN	count	number of elements in buffer (integer)	9			
IN	datatype	datatype of each buffer element (handle)	10			
OUT	status	status object (Status)	11			
			12			
int MPI_F	13					
	14					
MDT Eile	mond showed (the buf sour	at deteture status isomer) DIND(C)	¹⁵ ticket-248T.			
	MPI_File), INTENT(IN) ::	<pre>ht, datatype, status, ierror) BIND(C) fb</pre>	16			
	*), DIMENSION() :: bu		17			
	ER, INTENT(IN) :: count	_	18 19			
	MPI_Datatype), INTENT(IN)	:: datatype	20			
TYPE(21					
INTEG	22					
MPT FTIF	23					
	> BUF(*)	IT, DATATYPE, STATUS, IERROR)	24			
INTEG	25					
	26					
{void MPI	::File::Read_shared(void*		27			
	deprecated, see Section 1	datatype, MPI::Status& status) (binding				
	aeprecatea, see Section 1	<i>0.2)</i> {	29			
{void MPI	::File::Read_shared(void*		30 5 2) 31			
	const MPI::Datatype&	datatype) (binding deprecated, see Section 1	5.2) 32 32			
	}		33			
MPI_F	ILE_READ_SHARED reads a	file using the shared file pointer.	34			
			35			
	WRITE_SHARED(fh, buf, cou	nt datatura status)	36			
	× ×	,	37			
INOUT	fh	file handle (handle)	38			
IN	buf	initial address of buffer (choice)	39			
IN	count	number of elements in buffer (integer)	40			
IN	datatype	datatype of each buffer element (handle)	41 42			
			43			
OUT	status	status object (Status)	44			
			45 + i al so + 1.40			
int MPI_F		e fh, <mark>const</mark> void *buf, int count, e, MPI_Status *status)	$\frac{10}{46}$ ticket 140.			
MDT T#1	⁴⁷ ticket-248T.					
MPI_File_write_shared(fh, buf, count, datatype, status, ierror) BIND(C) 48						

```
1
                     TYPE(MPI_File), INTENT(IN) :: fh
           \mathbf{2}
                     TYPE(*), DIMENSION(...), INTENT(IN) :: buf
           3
                     INTEGER, INTENT(IN) :: count
           4
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           5
                     TYPE(MPI_Status) :: status
           6
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           7
                MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
           8
                     <type> BUF(*)
           9
                     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
           10
           11
                {void MPI::File::Write_shared(const void* buf, int count,
           12
                               const MPI::Datatype& datatype, MPI::Status& status) (binding
           13
                               deprecated, see Section 15.2 }
           14
                {void MPI::File::Write_shared(const void* buf, int count,
           15
                               const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
           16
                               ł
           17
           18
                    MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
           19
           20
                MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)
           21
           22
                  INOUT
                           fh
                                                       file handle (handle)
           23
                  OUT
                           buf
                                                       initial address of buffer (choice)
           24
                  IN
                           count
                                                       number of elements in buffer (integer)
           25
           26
                                                       datatype of each buffer element (handle)
                  IN
                           datatype
           27
                  OUT
                            request
                                                       request object (handle)
           28
           29
                int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
           30
                               MPI_Datatype datatype, MPI_Request *request)
           31
ticket-248T.
           32
                MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) BIND(C)
           33
                     TYPE(MPI_File), INTENT(IN) :: fh
           34
                     TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
           35
                     INTEGER, INTENT(IN) :: count
           36
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           37
                     TYPE(MPI_Request), INTENT(OUT) :: request
           38
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           39
                MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
           40
                     <type> BUF(*)
           41
                     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
           42
           43
                {MPI::Request MPI::File::Iread_shared(void* buf, int count,
           44
                               const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
           45
                               }
           46
                     MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
           47
                interface.
           48
```

MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)				
INOUT	fh	file handle (handle)	2	
IN	buf	initial address of buffer (choice)	4	
IN	count	number of elements in buffer (integer)	5	
IN	datatype	datatype of each buffer element (handle)	6	
OUT	request	request object (handle)	7 8	
		· · · · · · · · · · · · · · · · · · ·	9	
int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,				
MPI_Datatype datatype, MPI_Request *request)				
MPI_File_	_iwrite_shared(fh, buf, c	ount, datatype, request, ierror) BIND(C)	$_{12} \text{ ticket } 229.1.$ $_{13} \text{ ticket } 248 \text{T}.$	
	<pre>MPI_File), INTENT(IN) ::</pre>		14	
		T(IN), ASYNCHRONOUS :: buf	15	
	ER, INTENT(IN) :: count [MPI_Datatype), INTENT(IN		16	
	(MPI_Request), INTENT(OUT		17 18	
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	18	
MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)				
<pre><type> BUF(*)</type></pre>				
INTEG	ER FH, COUNT, DATATYPE,	REQUEST, IERROR	22	
{MPI::Rec	uest MPI:::File::Iwrite_s	hared(const void* buf, int count,	23 24	
	const MPI::Datatype&	: datatype) (binding deprecated, see Section 15.2)	25	

```
MPI_FILE_IWRITE_SHARED is a nonblocking version of the
MPI_FILE_WRITE_SHARED interface.
```

Collective Operations

}

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not require that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI_FILE_WRITE_ORDERED rather than MPI_FILE_WRITE_SHARED) may enable an implementation to optimize access, improving performance. (End of advice to users.)

26

27

2829 30

31

32

33

34

35

36

37

38

39

40 41

42

43

44

45

1 Advice to implementors. Accesses to the data requested by all processes do not have 2 to be serialized. Once all processes have issued their requests, locations within the file 3 for all accesses can be computed, and accesses can proceed independently from each 4 other, possibly in parallel. (End of advice to implementors.) 56 $\overline{7}$ MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status) 8 INOUT fh file handle (handle) 9 OUT buf initial address of buffer (choice) 10 11 IN count number of elements in buffer (integer) 12IN datatype datatype of each buffer element (handle) 13 OUT status status object (Status) 14 1516 int MPI_File_read_ordered(MPI_File fh, void *buf, int count, 17MPI_Datatype datatype, MPI_Status *status) ticket-248T. $_{18}$ MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) BIND(C) 19 TYPE(MPI_File), INTENT(IN) :: fh 20TYPE(*), DIMENSION(...) :: buf 21INTEGER, INTENT(IN) :: count 22 TYPE(MPI_Datatype), INTENT(IN) :: datatype 23TYPE(MPI_Status) :: status 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) 27INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 2829 {void MPI::File::Read_ordered(void* buf, int count, 30 const MPI::Datatype& datatype, MPI::Status& status) (binding 31 deprecated, see Section 15.2 } 32 33 {void MPI::File::Read_ordered(void* buf, int count, 34 const MPI::Datatype& datatype) (binding deprecated, see Section 15.2) 35 } 36 MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED 37 interface. 38 39 40MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status) 41 fh INOUT file handle (handle) 42IN buf initial address of buffer (choice) 43 44IN count number of elements in buffer (integer) 45IN datatype datatype of each buffer element (handle) 46 OUT status object (Status) status 47

int MPI_F		le fh, <mark>const</mark> void *buf, int count, be, MPI_Status *status)	$\frac{1}{2}$ ticket140.
TYPE(TYPE(INTEG TYPE(TYPE(<pre>write_ordered(fh, buf, c MPI_File), INTENT(IN) :: *), DIMENSION(), INTEN ER, INTENT(IN) :: count MPI_Datatype), INTENT(IN MPI_Status) :: status ER, OPTIONAL, INTENT(OUT</pre>	T(IN) :: buf) :: datatype	3 ticket-248T. 4 5 6 7 8 9 10
<type< td=""><td>> BUF(*)</td><td>OUNT, DATATYPE, STATUS, IERROR) STATUS(MPI_STATUS_SIZE), IERROR</td><td>11 12 13</td></type<>	> BUF(*)	OUNT, DATATYPE, STATUS, IERROR) STATUS(MPI_STATUS_SIZE), IERROR	11 12 13
{void MPI		nst void* buf, int count, z datatype, MPI::Status& status)(binding 15.2)}	14 15 16 17
{void MPI		nst void* buf, int count, a datatype)(binding deprecated, see Section 15.2)	18 19 20 21
MPI_F interface.	FILE_WRITE_ORDERED is a c	collective version of the $MPI_FILE_WRITE_SHARED$	21 22 23 24
to call the		specified when the file was opened, it is erroneous PI_FILE_SEEK_SHARED and	25 26 27 28 29 30
MPI_FILE_	_SEEK_SHARED(fh, offset, wh	nence)	31
INOUT	fh	file handle (handle)	32
IN	offset	file offset (integer)	33 34
IN	whence	update mode (state)	35 36
int MPI_F	lile_seek_shared(MPI_File	fh, MPI_Offset offset, int whence)	37
TYPE(seek_shared(fh, offset, MPI_File), INTENT(IN) :: ER(KIND=MPI_OFFSET_KIND)	fh	³⁸ ticket-248T. ³⁹ ⁴⁰
	ER, INTENT(IN) :: whenc ER, OPTIONAL, INTENT(OUT		42
			43 44
	SEEK_SHARED(FH, OFFSET,	WHENCE, IERROR)	45
	ER FH, WHENCE, IERROR ER(KIND=MPI_OFFSET_KIND)	OFFSET	46
11110		0.1.001	47
			48

	1 2	{void MP		_shared(MPI::: d, see Section 1		int whence) (bind	ling
	3 4 5		FILE_SEEK_SH	*	the shared file point	nter according to v	whence, which
	6	• MPI	_SEEK_SET: the	pointer is set to	o offset		
	7 8	• MPI	_SEEK_CUR: the	pointer is set to	o the current poin	nter position plus c	offset
	9 10	• MPI	_SEEK_END: the	pointer is set t	o the end of file p	lus offset	
	11			-	-	sses in the commu	nicator group
	12 13	associated	d with the file ha		, .	SHARED with th	• •
	14		and whence. offset can be neg	ative, which all	lows seeking backy	wards. It is errone	ous to seek to
	15 16	a negative position in the view.					
	17						
	18	MPI_FILE	E_GET_POSITIO	N_SHARED(fh,	offset)		
	19 20	IN	fh		file handle (handle	;)	
	20	OUT	offset		offset of shared poi	inter (integer)	
	22						、
ticket-248T.	23 • ₂₄	int MPL_	File_get_posi	tion_shared(M	Pl_File fh, MPl	_Offset *offset)
	24 25		· ·		ffset, ierror)	BIND(C)	
	26		CFP(KIND-MDT)		fh INTENT(OUT) ::	offect	
	27		GER, OPTIONAL			ollset	
	28			-	FFSET, IERROR)		
	29 30		GER FH, IERRO		FFSEI, IERRUR)		
	31		GER(KIND=MPI_		OFFSET		
	32	{MPT::Of	fset MPT::File	e::Get positi	on shared() con	st(binding deprece	ated see
	33	(Section 1	-			
	34 35	MPI	FILE GET POS	ITION SHARE) returns in offs	set, the current po	osition of the
	36				to the current vie	,	solution of the
	37						
	38					all to MPI_FILE_SE nt position. To set	
	39 40		0			ert offset into an	-
	41					call MPI_FILE_SE	=
	42	the	resulting displace	cement. (End of	f advice to users.)		
	43						
	44	13.4.5 S	Split Collective [Data Access Ro	utines		
	45 46					I/O operations fo	
	47		o .			outines are referred	-
	48	collective	routines becaus	e a single collec	tive operation is s	split in two: a begi	n routine and

test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc. Split collective data access operations on a file handle **fh** are subject to the semantic

Split collective data access operations on a file handle fh are subject to the semantic rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one process does not match an MPI_FILE_READ_ALL_BEGIN/ MPI_FILE_READ_ALL_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid [many (though not all) of]the problems described in "A Problem with Code Movements and Register Optimization," [Section 16.2.16, page 679.]Section 16.2.17 on page 680, but not all of the problems described in Section 16.2.16 on page 679.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```
MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);
```

is erroneous.

• In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify

Unofficial Draft for Comment Only

²⁸ ²⁹ ³⁰ ticket238-J. ³¹ ticket238-J.

ticket238-J.

```
3
4
```

1 2

5

6 7

8

9

10 11

12

13 14

15

16

17

18

19

20

21

22

23 24

25

26

27

32

33

34

35

36

37 38

39

40

41 42

43 44

 $45 \\ 46$

47

48

```
1
                       the implementation in the multithreaded case. (Note that we have already disallowed
            \mathbf{2}
                       having two threads begin a split collective operation on the same file handle since only
            3
                       one split collective operation can be active on a file handle at any time.)
            4
                      The arguments for these routines have the same meaning as for the equivalent collective
            5
                 versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and
            6
                 MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL).
            7
                 The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation
            8
                 that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END)
            9
            10
                 produces the result as defined for the equivalent collective routine (i.e.,
                 MPI_FILE_READ_ALL).
            11
                      For the purpose of consistency semantics (Section 13.6.1, page 567), a matched pair
            12
                 of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and
            13
                 MPI_FILE_READ_ALL_END) compose a single data access.
           14
            15
            16
                 MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
            17
                   IN
                             fh
            18
                                                         file handle (handle)
            19
                   IN
                             offset
                                                         file offset (integer)
           20
                   OUT
                             buf
                                                         initial address of buffer (choice)
           21
           22
                   IN
                             count
                                                         number of elements in buffer (integer)
           23
                   IN
                                                         datatype of each buffer element (handle)
                             datatype
           ^{24}
            25
                 int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
            26
                                 int count, MPI_Datatype datatype)
           27
ticket-248T.
            28
                 MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
           29
                                BIND(C)
           30
                      TYPE(MPI_File), INTENT(IN) :: fh
           31
                      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                                           offset
                      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
            32
           33
                      INTEGER, INTENT(IN) :: count
           34
                      TYPE(MPI_Datatype), INTENT(IN) :: datatype
           35
                      INTEGER, OPTIONAL, INTENT(OUT) ::
                                                              ierror
           36
                 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
           37
                      <type> BUF(*)
           38
                      INTEGER FH, COUNT, DATATYPE, IERROR
           39
                      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
            40
           ^{41}
                 {void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf,
           42
                                 int count, const MPI::Datatype& datatype) (binding deprecated, see
           43
                                 Section 15.2 }
           44
            45
            46
            47
            48
```

MPI_FI	LE_READ_AT_ALL_I	END(fh, buf, status)	1	
IN	fh	file handle (handle)	2 3	
OUT	buf	initial address of buffer (choice)	4	
OUT	status	status object (Status)	5	
			6	
int MP	I_File_read_at_al	l_end(MPI_File fh, void *buf, MPI_Status *status)		
MPT Fi	le read at all end	d(fh, buf, status, ierror) BIND(C)	⁸ ticket-248T.	
	PE(MPI_File), INT		10	
ТҮ	PE(*), DIMENSION(), ASYNCHRONOUS :: buf	11	
	PE(MPI_Status) ::		12	
IN	TEGER, OPTIONAL, I	INTENT(OUT) :: ierror	13	
MPI_FI	LE_READ_AT_ALL_ENI	D(FH, BUF, STATUS, IERROR)	14	
	ype> BUF(*)		15 16	
IN	TEGER FH, STATUS(I	MPI_STATUS_SIZE), IERROR	17	
{void	MPI::File::Read_at	t_all_end(void* buf, MPI::Status& status)(binding	18	
	deprecated,	see Section 15.2) }	19	
{void	MPI::File::Read_at	t_all_end(void* buf)(binding deprecated, see Section 15.2)	20 21	
-	}			
			22 23	
			24	
MPI_FI	LE_WRITE_AT_ALL_	_BEGIN(fh, offset, buf, count, datatype)	25	
INOU	T fh	file handle (handle)	26	
IN	offset	file offset (integer)	27 28	
IN	buf	initial address of buffer (choice)	29	
IN	count	number of elements in buffer (integer)	30	
IN	datatype	datatype of each buffer element (handle)	31	
			32	
int MP	I_File_write_at_a	ll_begin(MPI_File fh, MPI_Offset offset, const	$^{33}_{34}$ ticket 140.	
	void *buf,	int count, MPI_Datatype datatype)		
MPI Fi	le write at all be	egin(fh, offset, buf, count, datatype, ierror)	$^{35}_{_{36}}$ ticket-248T.	
-	BIND(C)		37	
ТҮ	PE(MPI_File), INT	ENT(IN) :: fh	38	
		FSET_KIND), INTENT(IN) :: offset	39	
), INTENT(IN), ASYNCHRONOUS :: buf	40 41	
	TEGER, INTENT(IN) PE(MPI Datatype)	INTENT(IN) :: datatype	42	
	• •	INTENT(OUT) :: ierror	43	
		EGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)	44	
	vpe> BUF(*)	LGIN(FII, OFFSEI, BOF, COUNI, DAIAIIFE, IERROR)	45	
	TEGER FH, COUNT, I	DATATYPE, IERROR	46 47	
IN	TEGER(KIND=MPI_OF	FSET_KIND) OFFSET	48	

```
1
                 {void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
            \mathbf{2}
                                int count, const MPI::Datatype& datatype) (binding deprecated, see
            3
                                Section 15.2 }
            4
            5
            6
                 MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
            7
                   INOUT
                            fh
                                                        file handle (handle)
            8
                            buf
            9
                   IN
                                                        initial address of buffer (choice)
           10
                   OUT
                            status
                                                        status object (Status)
           11
           12
  ticket 140.
                 int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
           13
                                MPI_Status *status)
ticket-248T.<sup>14</sup>
           15
                 MPI_File_write_at_all_end(fh, buf, status, ierror) BIND(C)
           16
                     TYPE(MPI_File), INTENT(IN) :: fh
           17
                     TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
           18
                     TYPE(MPI_Status) :: status
           19
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           20
                 MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
           21
                     <type> BUF(*)
           22
                     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
           23
           24
                 {void MPI::File::Write_at_all_end(const void* buf,
           25
                                MPI::Status& status) (binding deprecated, see Section 15.2) }
           26
                 {void MPI::File::Write_at_all_end(const void* buf) (binding deprecated, see
           27
                                Section 15.2 }
           28
           29
           30
                 MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)
           ^{31}
           32
                   INOUT
                            fh
                                                        file handle (handle)
           33
                   OUT
                            buf
                                                        initial address of buffer (choice)
           34
                   IN
                            count
                                                        number of elements in buffer (integer)
           35
           36
                   IN
                            datatype
                                                        datatype of each buffer element (handle)
           37
           38
                 int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
           39
                                MPI_Datatype datatype)
ticket-248T. ^{40}
                 MPI_File_read_all_begin(fh, buf, count, datatype, ierror) BIND(C)
           41
           42
                     TYPE(MPI_File), INTENT(IN) :: fh
                     TYPE(*), DIMENSION(..), ASYNCHRONOUS ::
                                                                    buf
           43
                     INTEGER, INTENT(IN) :: count
           44
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           45
                     INTEGER, OPTIONAL, INTENT(OUT) ::
           46
                                                             ierror
           47
                 MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
           48
```

<type INTEG</type 	1 2		
{void MPI	3 4 5		
	}		6
			7 8
MPI_FILE_	9		
INOUT	fh	file handle (handle)	10
OUT	buf	initial address of buffer (choice)	11
OUT	status	status object (Status)	12
001	Status	status object (Status)	13 14
		e fh, void *buf, MPI_Status *status)	$^{15}_{16}$ ticket-248T.
	read_all_end(fh, buf, sta		17
	<pre>MPI_File), INTENT(IN) :: *), DIMENSION(), ASYNCH</pre>		18
	MPI_Status) :: status		19 20
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	20
MPI_FILE_	22		
	> BUF(*)	,	23
INTEG	ER FH, STATUS(MPI_STATUS	_SIZE), IERROR	24
{void MPI	::File::Read_all_end(void	l* buf, MPI:::Status& status)(binding	25 26
C C	deprecated, see Section 1	· –	27
{void MPI	::File::Read all end(void	<pre>l* buf)(binding deprecated, see Section 15.2) }</pre>	28
(29
	30		
MPI_FILE_	WRITE_ALL_BEGIN(fh, buf, o	count, datatype)	31 32
INOUT	fh	file handle (handle)	33
IN	buf	initial address of buffer (choice)	34
IN	count	number of elements in buffer (integer)	35
IN	datatype	datatype of each buffer element (handle)	36
IIN	datatype	datatype of each build ciclicit (handle)	37 38
int MPI_F	ile_write_all_begin(MPI_F	File fh, const void *buf, int count,	³⁹ ticket140.
_	40		
	<pre>count, datatype, ierror) BIND(C) fb</pre>	$_{41}$ ticket-248T.	
	<pre>MPI_File), INTENT(IN) :: *). DIMENSION(). INTENT</pre>	IN F(IN), ASYNCHRONOUS :: buf	43
	ER, INTENT(IN) :: count		44
TYPE(MPI_Datatype), INTENT(IN)		45 46
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	47
MPI_FILE_	48		

```
1
                     <type> BUF(*)
            \mathbf{2}
                     INTEGER FH, COUNT, DATATYPE, IERROR
            3
                 {void MPI::File::Write_all_begin(const void* buf, int count,
            4
                                const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
            5
                                }
            6
            7
            8
                 MPI_FILE_WRITE_ALL_END(fh, buf, status)
            9
            10
                   INOUT
                            fh
                                                         file handle (handle)
           11
                   IN
                             buf
                                                        initial address of buffer (choice)
           12
                   OUT
                            status
                                                        status object (Status)
           13
           14
  ticket140. 15
                 int MPI_File_write_all_end(MPI_File fh, const void *buf,
            16
                                MPI_Status *status)
ticket-248T. 17
                 MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)
            18
                     TYPE(MPI_File), INTENT(IN) :: fh
           19
                     TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
           20
                     TYPE(MPI_Status) :: status
           21
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           22
           23
                 MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
           24
                      <type> BUF(*)
           25
                     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
           26
                 {void MPI::File::Write_all_end(const void* buf, MPI::Status& status)(binding
           27
                                deprecated, see Section 15.2 }
           28
           29
                 {void MPI::File::Write_all_end(const void* buf)(binding deprecated, see
           30
                                Section 15.2 }
           ^{31}
           32
           33
                 MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)
           34
                   INOUT
                            fh
                                                        file handle (handle)
           35
           36
                   OUT
                             buf
                                                        initial address of buffer (choice)
           37
                   IN
                                                        number of elements in buffer (integer)
                            count
           38
                   IN
                            datatype
                                                        datatype of each buffer element (handle)
           39
            40
                 int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
           41
           42
                                MPI_Datatype datatype)
ticket-248
T<br/>. _{\rm 43}
                 MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
           44
                     TYPE(MPI_File), INTENT(IN) :: fh
           45
                     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
            46
                     INTEGER, INTENT(IN) :: count
            47
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
           48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         \mathbf{2}
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
                                                                                         4
    INTEGER FH, COUNT, DATATYPE, IERROR
                                                                                         5
                                                                                         6
{void MPI::File::Read_ordered_begin(void* buf, int count,
                                                                                         7
               const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
                                                                                         8
               }
                                                                                         9
                                                                                         10
                                                                                         11
MPI_FILE_READ_ORDERED_END(fh, buf, status)
                                                                                         12
 INOUT
           fh
                                       file handle (handle)
                                                                                         13
 OUT
           buf
                                       initial address of buffer (choice)
                                                                                         14
                                                                                         15
 OUT
           status
                                       status object (Status)
                                                                                         16
                                                                                         17
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
                                                                                         ^{18} ticket-248T.
                                                                                         19
MPI_File_read_ordered_end(fh, buf, status, ierror) BIND(C)
                                                                                         20
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                         21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                         22
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         23
                                                                                         ^{24}
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
                                                                                         25
    <type> BUF(*)
                                                                                         26
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                         27
                                                                                         28
{void MPI::File::Read_ordered_end(void* buf, MPI::Status& status) (binding
                                                                                         29
               deprecated, see Section 15.2 }
                                                                                         30
{void MPI::File::Read_ordered_end(void* buf)(binding deprecated, see Section 15.2)
                                                                                         31
               }
                                                                                         32
                                                                                         33
                                                                                         34
MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
                                                                                         35
                                                                                         36
 INOUT
           fh
                                       file handle (handle)
                                                                                         37
 IN
           buf
                                       initial address of buffer (choice)
                                                                                         38
 IN
           count
                                       number of elements in buffer (integer)
                                                                                         39
                                                                                         40
 IN
           datatype
                                       datatype of each buffer element (handle)
                                                                                         41
                                                                                         42
                                                                                           ticket140.
int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
                                                                                         43
              MPI_Datatype datatype)
                                                                                         ^{44} ticket-248T.
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
                                                                                         45
                                                                                         46
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                         47
    INTEGER, INTENT(IN) :: count
                                                                                         48
```

42

```
1
                     TYPE(MPI_Datatype), INTENT(IN) :: datatype
            2
                     INTEGER, OPTIONAL, INTENT(OUT) ::
                                                             ierror
            3
                 MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
            4
                      <type> BUF(*)
            5
                     INTEGER FH, COUNT, DATATYPE, IERROR
            6
            7
                 {void MPI::File::Write_ordered_begin(const void* buf, int count,
            8
                                const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
            9
                                ł
           10
           11
           12
                 MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
           13
                   INOUT
                             fh
                                                         file handle (handle)
           14
           15
                   IN
                             buf
                                                         initial address of buffer (choice)
           16
                   OUT
                            status
                                                         status object (Status)
           17
           18
  ticket140. 19
                 int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
                                MPI_Status *status)
ticket-248T.<sup>20</sup>
           21
                 MPI_File_write_ordered_end(fh, buf, status, ierror) BIND(C)
           22
                     TYPE(MPI_File), INTENT(IN) :: fh
           23
                     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
           24
                     TYPE(MPI_Status) :: status
           25
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           26
                 MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
           27
                      <type> BUF(*)
           28
                     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
           29
           30
                 {void MPI::File::Write_ordered_end(const void* buf,
           ^{31}
                                MPI::Status& status) (binding deprecated, see Section 15.2) }
           32
                 {void MPI::File::Write_ordered_end(const void* buf) (binding deprecated, see
           33
                                Section 15.2 }
           34
           35
           36
                 13.5
                         File Interoperability
           37
           38
                 At the most basic level, file interoperability is the ability to read the information previously
           39
                 written to a file—not just the bits of data, but the actual information the bits represent.
           40
                 MPI guarantees full interoperability within a single MPI environment, and supports in-
           41
                 creased interoperability outside that environment through the external data representation
```

(Section 13.5.2, page 560) as well as the data conversion functions (Section 13.5.3, page 561). 43 Interoperability within a single MPI environment (which could be considered "oper-44ability") ensures that file data written by one MPI process can be read by any other MPI 45 process, subject to the consistency constraints (see Section 13.6.1, page 567), provided that 46 it would have been possible to start the two processes simultaneously and have them reside 4748

in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written. This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility that transfers and converts files between the native file system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. In a high-quality implementation of MPI, users will be able to manipulate MPI files using the same or similar tools that the native file system offers for manipulating its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports multiple data representations: "native," "internal," and "external32." An implementation may support additional data representations. MPI also supports userdefined data representations (see Section 13.5.3, page 561). The "native" and "internal" data representations are implementation dependent, while the "external32" representation is common to all MPI implementations and facilitates file interoperability. The data representation is specified in the *datarep* argument to MPI_FILE_SET_VIEW.

Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (End of advice to users.)

"native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

> Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (End of advice to users.)

Unofficial Draft for Comment Only

56

1 2

3

4

7 8

9

10 11

12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35 36

37

38

39

40 41

42

43

44

4546

47

1	Advice to implementors. When implementing read and write operations on
2	top of MPI message-passing, the message data should be typed as MPI_BYTE
3	to ensure that the message routines do not perform any type conversions on the
4	data. (End of advice to implementors.)
5	
6	"internal" This data representation can be used for I/O operations in a homogeneous
7	or heterogeneous environment; the implementation will perform type conversions if
8	necessary. The implementation is free to store data in any format of its choice, with
9	the restriction that it will maintain constant extents for all predefined datatypes in any
10	one file. The environment in which the resulting file can be reused is implementation-
11	defined and must be documented by the implementation.
12	defined and mast be documented by the implementation.
13	Rationale. This data representation allows the implementation to perform I/O
14	efficiently in a heterogeneous environment, though with implementation-defined
15	restrictions on how the file can be reused. (<i>End of rationale.</i>)
16	
17	Advice to implementors. Since "external32" is a superset of the functionality
18	provided by "internal," an implementation may choose to implement "internal"
19	as "external32." (End of advice to implementors.)
20	
21	"external32" This data representation states that read and write operations convert all
22	data from and to the "external32" representation defined in Section 13.5.2, page 560.
22	The data conversion rules for communication also apply to these conversions (see
24	Section 3.3.2, page 25-27, of the MPI-1 document). The data on the storage medium
24 25	is always in this canonical representation, and the data in memory is always in the
26	local process's native representation.
20	This data representation has several advantages. First, all processes reading the file
28	in a heterogeneous MPI environment will automatically have the data converted to
29	their respective native representations. Second, the file can be exported from one MPI
30	environment and imported into any other MPI environment with the guarantee that
31	the second environment will be able to read all the data in the file.
32	
33	The disadvantage of this data representation is that data precision and I/O perfor-
34	mance may be lost in data type conversions.
35	
36	Advice to implementors. When implementing read and write operations on top
	of MPI message-passing, the message data should be converted to and from the
37	"external32" representation in the client, and sent as type MPI_BYTE. This will
38 20	avoid possible double data type conversions and the associated further loss of
39	precision and performance. (End of advice to implementors.)
40	
41	13.5.1 Datatypes for File Interoperability
42	If the file data representation is other than "native," care must be taken in constructing
43	etypes and filetypes. Any of the datatype constructor functions may be used; however,
44 45	for those functions that accept displacements in bytes, the displacements must be specified
45 46	in terms of their values in the file for the file data representation being used. MPI will
40	interpret these byte displacements as is; no scaling will be done. The function

- ⁴⁷ interpret these byte displacements as is; no scaling will be done. The function
- ⁴⁸ MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the

dependent.

file. For etypes and filetypes that are portable datatypes (see Section 2.4, page 11), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype 12and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. The 14routine MPI_FILE_GET_FILE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user defined data representations. Otherwise, the etype and filetype must be constructed so that their typemap and extent are the same on any architecture. This can be achieved if they have an explicit upper bound and lower bound (defined either using MPI_LB and 20MPI_UB markers, or using MPI_TYPE_CREATE_RESIZED). This condition must also 21be fulfilled by any datatype that is used in the construction of the etype and filetype, 22if this datatype is replicated contiguously, either explicitly, by a call to 23MPI_TYPE_CONTIGUOUS, or implicitly, by a blocklength argument that is greater than one. If an etype or filetype is not portable, and has a typemap or extent that is architecture dependent, then the data layout specified by it on a file is implementation

File data representations other than "native" may be different from corresponding data representations in memory. Therefore, for these file data representations, it is important not to use hardwired byte offsets for file positioning, including the initial displacement that specifies the view. When a portable datatype (see Section 2.4, page 11) is used in a data access operation, any holes in the datatype are scaled to match the data representation. However, note that this technique only works when all the processes that created the file view build their etypes from the same predefined datatypes. For example, if one process uses an etype built from MPI_INT and another uses an etype built from MPI_FLOAT, the resulting views may be nonportable because the relative sizes of these types may differ from one data representation to another. (End of advice to users.)

39 40 41

		•	
IN	fh		file handle (handle)
IN	datatype		datatype (handle)
OUT	extent		datatype extent (integer)

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)

1

 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8

9

10

11

13

15

16

17

18

19

 24

25

26

2728

29

30

 31

32

33

34

35

36

37

|--|

ticket-248T. 3	<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, MPI_Aint *extent)</pre>
4 5 6 7 8	<pre>MPI_File_get_type_extent(fh, datatype, extent, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
9 10 11	MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR) INTEGER FH, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
12 13 14	<pre>{MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype)</pre>
15 16 17 18 19	Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 13.5.3, page 561), MPI uses the dtype_file_extent_fn callback to calculate the extent.
20 21 22 23 24	Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the prede- fined datatypes in this derived datatype using dtype_file_extent_fn (see Section 13.5.3, page 561). (End of advice to implementors.)
25 26	13.5.2 External Data Representation: "external32"
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required. All floating point values are in big-endian IEEE format [36] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous to the "Double" format. All integral values are in two's complement big-endian format. Big-endian means most significant byte at lowest address byte. For C _Bool, Fortran LOGICAL and C++ bool, 0 implies false and nonzero implies true. C float _Complex, double _Complex and long double _Complex as well as Fortran COMPLEX and DOUBLE COMPLEX are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [37]. Wide characters (of type MPI_WCHAR) are in Unicode format [60]. All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary
43 44 45 46	parts at the most significant bit of each part.According to IEEE specifications [36], the "NaN" (not a number) is system dependent.It should not be interpreted within MPI as anything other than "NaN."
47 48	Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR (see ftp://ds.internic.net/rfc/rfc1832.txt). (End of advice to implementors.)

	data is byte aligned, regardle (if the file view is contiguous)	ess of type. All data items are stored contiguously in	1 2	
1110 1110	(3	
A	dvice to implementors. All by	tes of LOGICAL and bool must be checked to determine	4	
$^{\mathrm{th}}$	e value. (End of advice to im	plementors.)	5	
			6	
		IPI_PACKED is treated as bytes and is not converted.	7	
		MPI_PACK has the option of placing a header in the	8	
be	eginning of the pack buffer. (A	End of advice to users.)	9	
Th	a size of the predefined datat	ypes returned from MPI_TYPE_CREATE_F90_REAL,	10	
MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined				
	in Section 16.2.9, page 666.		12	
III Decti	in 10.2.9, page 000.		13	
A	dvice to implementors. Wh	nen converting a larger size integer to a smaller size	14	
in	teger, only the less significant	bytes are moved. Care must be taken to preserve the	15	
sig	gn bit value. This allows no o	conversion errors if the data range is within the range	16	
of	the smaller size integer. $(En$	d of advice to implementors.)	17	
			18	
Tal	ble 13.2 specifies the sizes of p	predefined datatypes in "external32" format.	19	
			20	
13.5.3	User-Defined Data Represer	ntations	21	
There a	re two situations that cannot	be handled by the required representations:	22	
i nere a	te two situations that cannot	be handled by the required representations.	23	
1. a	user wants to write a file in a	representation unknown to the implementation, and	24 25	
2	user wents to read a file writte	en in a representation unknown to the implementation.	26	
2. a	user wants to read a me writte	in in a representation unknown to the implementation.	27	
Use	er-defined data representation	is allow the user to insert a third party converter into	28	
the I/O	stream to do the data repres	entation conversion.	29	
			30 31	
WIPI_RE		read_conversion_fn, write_conversion_fn,	32	
	dtype_file_extent_fn,	extra_state)	33	
IN	datarep	data representation identifier (string)	34	
IN	read_conversion_fn	function invoked to convert from file representation to	35	
		native representation (function)	36	
IN	write_conversion_fn	function invoked to convert from native representation	37	
IIN	write_conversion_m	to file representation (function)	38	
			39	
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as	40	
		represented in the file (function)	41	
IN extra_state extra state 42				
			43	
int MPI	[_Register_datarep(const	char *datarep,	44 ticket 140.	
		rsion function *read conversion fn	45	

Туре	Length	Optional Type	Length
MPI_PACKED	1	MPI_INTEGER1	
MPI_BYTE	1	MPI_INTEGER2	2
MPI_CHAR	1	MPI_INTEGER4	4
MPI_UNSIGNED_CHAR	1	MPI_INTEGER8	8
MPI_SIGNED_CHAR	1	MPI_INTEGER16	16
MPI_WCHAR	2		
MPI_SHORT	2	MPI_REAL2	2
MPI_UNSIGNED_SHORT	2	MPI_REAL4	4
MPI_INT	4	MPI_REAL8	8
MPI_UNSIGNED	4	MPI_REAL16	16
MPI_LONG	4		
 MPI_UNSIGNED_LONG	4	MPI_COMPLEX4	2*2
MPI_LONG_LONG_INT	8	MPI_COMPLEX8	2*4
MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16	2*8
MPI_FLOAT	4	MPI_COMPLEX32	2*16
MPI_DOUBLE	8		2.10
MPI_LONG_DOUBLE	16		
	10		
MPI_C_BOOL	[ticke	t171.][4]1	
MPI_INT8_T	1		
MPI_INT16_T	2		
MPI_INT32_T	4		
MPI_INT64_T	8		
MPI_UINT8_T	1		
MPI_UINT16_T	2		
MPI_UINT32_T	4		
	4 8		
MPI_UINT64_T MPI_AINT	8		
MPI_OFFSET	8		
MPI_C_COMPLEX	2*4		
MPI_C_FLOAT_COMPLEX	2*4		
MPI_C_DOUBLE_COMPLEX	2*8		
MPI_C_LONG_DOUBLE_COMPLEX	2*16		
MPI_CHARACTER	1		
MPI_LOGICAL	4		
MPI_INTEGER	4		
MPI_REAL	4		
MPI_DOUBLE_PRECISION	8		
MPI_COMPLEX	2*4		
MPI_DOUBLE_COMPLEX	2*8		
Table 19.0. 6	ortomalon"	gizes of prodofined datation	
Table 13.2:	external52"	sizes of predefined datatypes	

	1 ticket-248T.
<pre>MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,</pre>	2
<pre>dtype_file_extent_fn, extra_state, ierror) BIND(C)</pre>	3
CHARACTER(LEN=*), INTENT(IN) :: datarep	4
<pre>PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn</pre>	5
<pre>PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn</pre>	6
<pre>PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn</pre>	7
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
MOT DECISTED DATADED (DATADED DEAD CONVEDSION EN UDITE CONVEDSION EN	10
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,	11
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)	12
CHARACTER*(*) DATAREP	13
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN	14
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	15
INTEGER IERROR	16
<pre>{void MPI::Register_datarep(const char* datarep,</pre>	17
MPI::Datarep_conversion_function* read_conversion_fn,	18
<pre>MPI::Datarep_conversion_function* write_conversion_fn,</pre>	19
<pre>MPI::Datarep_extent_function* dtype_file_extent_fn,</pre>	20
<pre>void* extra_state)(binding deprecated, see Section 15.2) }</pre>	21

The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn with the data representation identifier datarep. datarep can then be used as an argument to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. MPI_REGISTER_DATAREP is a local operation and only registers the data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler (see Section 13.7, page 576). The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

³⁸ ticket229.1. ³⁹ ticket-248T.

```
1
                 {typedef void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
            \mathbf{2}
                                MPI:::Aint& file_extent, void* extra_state); (binding deprecated,
            3
                                see Section 15.2
            4
                     The function dtype_file_extent_fn must return, in file_extent, the number of bytes re-
            5
                 quired to store datatype in the file representation. The function is passed, in extra_state,
            6
                 the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call
            7
                 this routine with predefined datatypes employed by the user.
            8
            9
                 Datarep Conversion Functions
           10
           11
                 typedef int MPI_Datarep_conversion_function(void *userbuf,
           12
                                MPI_Datatype datatype, int count, void *filebuf,
           13
                                MPI_Offset position, void *extra_state);
ticket-248T. 14
                 ABSTRACT INTERFACE
           15
                   SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
           16
                   filebuf, position, extra_state, ierror) BIND(C)
           17
                        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
           18
                        TYPE(C_PTR), VALUE :: userbuf, filebuf
           19
                        TYPE(MPI_Datatype) :: datatype
           20
                        INTEGER :: count, ierror
           21
                        INTEGER(KIND=MPI_OFFSET_KIND) :: position
           22
                        INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
           23
           24
                 SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
           25
                                POSITION, EXTRA_STATE, IERROR)
           26
                     <TYPE> USERBUF(*), FILEBUF(*)
           27
                     INTEGER COUNT, DATATYPE, IERROR
           28
                     INTEGER(KIND=MPI_OFFSET_KIND) POSITION
           29
                     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
           30
                 {typedef void MPI::Datarep_conversion_function(void* userbuf,
           ^{31}
                                MPI::Datatype& datatype, int count, void* filebuf,
           32
                                MPI::Offset position, void* extra_state); (binding deprecated, see
           33
                                Section 15.2
           34
           35
                     The function read_conversion_fn must convert from file data representation to native
           36
                 representation. Before calling this routine, MPI allocates and fills filebuf with
           37
                 count contiguous data items. The type of each data item matches the corresponding entry
           38
                 for the predefined datatype in the type signature of datatype. The function is passed, in
           39
                 extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The
           40
                 function must copy all count data items from filebuf to userbuf in the distribution described
           41
                 by datatype, converting each data item from file representation to native representation.
           42
                 datatype will be equivalent to the datatype that the user passed to the read function. If the
           43
                 size of datatype is less than the size of the count data items, the conversion function must
           44
                 treat datatype as being contiguously tiled over the userbuf. The conversion function must
           45
                 begin storing converted data at the location in userbuf specified by position into the (tiled)
           46
                 datatype.
           47
```

48

Advice to users. Although the conversion functions have similarities to MPI_PACK

and MPI_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

Advice to implementors. A converted read operation could be implemented as follows:

- 1. Get file extent of all data items
- 2. Allocate a filebuf large enough to hold all count data items
- 3. Read data from file into filebuf
- 4. Call read_conversion_fn to convert data and place it into userbuf
- 5. Deallocate filebuf

(End of advice to implementors.)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the

Unofficial Draft for Comment Only

1

2

3

4

5

6

7 8

9 10

11

12

13

14 15

16 17

18

19

20

21

22

23

 24

25 26

27

28

29

30

 31

32 33

34

35

36

37 38

39

40

41

42

43

44

45

46

47

write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant $MPI_CONVERSION_FN_NULL$ may be used as either

write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke
 write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data
 access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by
 using a filebuf large enough to hold all the requested data items or else by making repeated
 calls to the conversion function with the same datatype argument and appropriate values
 for position.

An implementation will only invoke the callback routines in this section

(read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 13.4, page 527, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dtype_file_extent_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are
 restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion
 functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has
 a value other than MPI_SUCCESS, the implementation will raise an error in the class
 MPI_ERR_CONVERSION.

 $23 \\ 24$

25

35

36

37

38 39

40

41

42

43

44 45

46

47 48

13.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is used. Compatibility is guaranteed using "external32" provided at least one of the following conditions is met.

- The data access routines directly use types enumerated in Section 13.5.2, page 560, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 16.2.9, page 662).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatiblity with another implementation's "native" or "internal" representation.

1

 $\mathbf{2}$

3

Advice to users. Section 16.2.9, page 662, defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

13.6 Consistency and Semantics

13.6.1File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. Userimposed consistency may be obtained using program order and calls to MPI_FILE_SYNC.

Let FH_1 be the set of file handles created from one particular collective open of the file FOO, and FH_2 be the set of file handles created from a different collective open of FOO. Note that nothing restrictive is said about FH_1 and FH_2 : the sizes of FH_1 and 20 FH_2 may be different, the groups of processes used for each open may or may not intersect, 21the file handles in FH_1 may be destroyed before those in FH_2 are created, etc. Consider 22the following three cases: a single file handle (e.g., $fh_1 \in FH_1$), two file handles created 23from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$).

For the purpose of consistency semantics, a matched pair (Section 13.4.5, page 548) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access operations.

Advice to users. For an MPI_FILE_IREAD and MPI_WAIT pair, the operation begins when MPI_FILE_IREAD is called and ends when MPI_WAIT returns. (End of advice to users.)

Assume that A_1 and A_2 are two data access operations. Let D_1 (D_2) be the set of absolute byte displacements of every byte accessed in A_1 (A_2). The two data accesses overlap if $D_1 \cap D_2 \neq \emptyset$. The two data accesses *conflict* if they overlap and at least one is a write access.

Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) SEQ_{fh} is a "write sequence" if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, SEQ_1 and SEQ_2 , we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

Unofficial Draft for Comment Only

1

2

3

4 5

> 6 7

8

9

10

11

12

13

14

15

16

17

18

19

 24

25

26

27

28

29

30

31

32 33

34

35

36 37

38

39

40

41

42

43

44

45

46

47

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

 $\mathbf{5}$ Case 1: $fh_1 \in FH_1$ All operations on fh_1 are sequentially consistent if atomic mode is 6 set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume A_1 is a data access operation using fh_{1a} , and A_2 is a data access operation using fh_{1b} . If for any access A_1 , there is no access A_2 that conflicts with A_1 , then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

17Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from 18 distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC 19must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC). 20

Sequential consistency is guaranteed among accesses to a single file if for any write sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file which is *concurrent* with SEQ_1 . To guarantee sequential consistency when there are write sequences,

MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 13.6.10, page 572, for further clarification of some of these consistency semantics.

MPI_FILE_SET_ATOMICITY(fh, flag)

	30 31	INOUT	fh	file handle (handle)		
ticket-248T.	32	IN	flag	true to set atomic mode, $false$ to set nonatomic mode		
	33			(logical)		
	34					
	35	<pre>int MPI_File_set_atomicity(MPI_File fh, int flag)</pre>				
		MPI_File_set_atomicity(fh, flag, ierror) BIND(C)				
	37	TYPE(MPI_File), INTENT(IN) :: fh				
	38					
	39	LOGICAL, INTENT(IN) :: flag				
	40	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror		
	41	MPI_FILE_S	SET_ATOMICITY(FH, FLAG, I	ERROR)		
	42	INTEGER FH, IERROR				
	43	LOGIC	AL FLAG			
	44	(
	45	{void MPI	::File::Set_atomicity(boo	ol flag)(binding deprecated, see Section 15.2) }		
	46	Let FH be the set of file handles created by one collective open. The consister semantics for data access operations using FH is set by collectively calling				
	47					
	48					

1

 $\mathbf{2}$

3

4

 $\overline{7}$

8 9

10

11

12

13

14

15

16

21

22

23

 24

25

26

MPI_FILE_SET_ATOMICITY on *FH*. MPI_FILE_SET_ATOMICITY is collective; all processes in the group must pass identical values for fh and flag. If flag is true, atomic mode is set; if flag is false, nonatomic mode is set.

Changing the consistency semantics for an open file only affects new data accesses. All completed data accesses are guaranteed to abide by the consistency semantics in effect during their execution. Nonblocking data accesses and split collective operations that have not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode consistency semantics.

Advice to implementors. Since the semantics guaranteed by atomic mode are stronger than those guaranteed by nonatomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (End of advice to implementors.)

MPI_FILE_GET_ATOMICITY(fh, flag)

			17			
IN	fh	file handle (handle)	18			
OUT	flag	true if atomic mode, false if nonatomic mode (le	ogical) ¹⁹			
	-		20			
int MPI	<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>					
	<pre>MPI_File_get_atomicity(fh, flag, ierror) BIND(C)</pre>					
TYPI	TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag					
INTE	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
MPI_FIL	MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)					
	EGER FH, IER		28			
	ICAL FLAG		29			
			30			

```
{bool MPI::File::Get_atomicity() const(binding deprecated, see Section 15.2) }
```

MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled.

	00				
MPI_FILE_SYNC(fh)	37				
INOUT fh	file handle (handle)	38			
	me nandie (nandie)	39			
int MDI File and (MDI File fb)		40			
<pre>int MPI_File_sync(MPI_File fh)</pre>					
MPI_File_sync(fh, ierror) BIND(C)					
<pre>TYPE(MPI_File), INTENT(IN) ::</pre>	fh	43			
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	44			
NOT FILE GUNG(FIL TEDDOD)		45			
MPI_FILE_SYNC(FH, IERROR)	46				
INTEGER FH, IERROR		47			
<pre>{void MPI::File::Sync()(binding deprecated, see Section 15.2)}</pre>					

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 7

8

9

10

11

12

31 32

33

34

¹ Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process ² to be transferred to the storage device. If other processes have made updates to the storage ³ device, then all such updates become visible to subsequent reads of fh by the calling process. ⁴ MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see ⁵ above).

6 N

MPI_FILE_SYNC is a collective operation.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SYNC—otherwise, the call to MPI_FILE_SYNC is erroneous.

9 10 11

12

7

8

13.6.2 Random Access vs. Sequential Files

MPI distinguishes ordinary random access files from sequential stream files, such as pipes 13 and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL 14flag set in the amode. For these files, the only permitted data access operations are shared 15file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the 16notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and 17MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified 18 for the data access routines do not apply. The amount of data accessed by a data access 19 operation will be the amount requested unless the end of file is reached or an error is raised. 20

21 22 23

24

Rationale. This implies that reading on a pipe will always wait until the requested amount of data is available or until the process writing to the pipe has issued an end of file. (*End of rationale.*)

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI_FILE_SET_SIZE with size set to the current position) followed by the write.

29 30

 31

32

33

13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

45

46

13.6.4 Collective File Operations

⁴⁷ Collective file operations are subject to the same restrictions as collective communication
 ⁴⁸ operations. For a complete discussion, please refer to the semantics set forth in Section 5.13

on page 228.

Collective file operations are collective over a dup of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

13.6.5 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

13.6.6 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

13.6.7 MPI_Offset Type

MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset.

In Fortran, the corresponding integer is an integer [of kind]with kind parameter MPI_OFFSET_KIND, [defined in mpif.h and the mpi module]which is defined in the mpi_f08 module, the mpi module and the mpif.h include file.

In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 16.3, page 694).

13.6.8 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

³⁶ ticket230-B.

³⁷ ticket230-B.

¹ portability. However, there are still cases where some information may be necessary to ² optimize file layout. This information can be provided as *hints* specified via *info* when a file ³ is created (see Section 13.2.8, page 521).

13.6.9 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI_FILE_SET_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI_FILE_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.

When applying consistency semantics, calls to MPI_FILE_SET_SIZE and

MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and
 MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).

21 22 23

 24

25

26

27

28

Advice to users. Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 13.6.1, page 567, are satisfied. (End of advice to users.)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

29 30 31

32

33

34

35

36 37

38

42 43

44

Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

13.6.10 Examples

The examples in this section illustrate the application of the MPI consistency and semantics
 guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of b will be 5. If nonatomic mode is set, the results of the read are undefined.

Unofficial Draft for Comment Only

4 5

6

7

8

9

10

11

12

13 14

15

16 17

18

19

```
1
/* Process 0 */
                                                                                      2
int i, a[10];
                                                                                      3
int TRUE = 1;
                                                                                      4
for ( i=0;i<10;i++)
                                                                                      5
                                                                                      6
   a[i] = 5;
                                                                                      7
                                                                                      8
MPI_File_open( MPI_COMM_WORLD, "workfile",
                MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
                                                                                      9
                                                                                      10
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                      11
MPI_File_set_atomicity( fh0, TRUE ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status) ;
                                                                                      12
/* MPI_Barrier( MPI_COMM_WORLD ) ; */
                                                                                      13
                                                                                      14
/* Process 1 */
                                                                                      15
int b[10];
                                                                                      16
int TRUE = 1;
                                                                                      17
MPI_File_open( MPI_COMM_WORLD, "workfile",
                                                                                      18
                MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
                                                                                      19
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                      20
MPI_File_set_atomicity( fh1, TRUE ) ;
                                                                                      21
/* MPI_Barrier( MPI_COMM_WORLD ) ; */
                                                                                      22
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status) ;
                                                                                      23
                                                                                      ^{24}
A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
                                                                                      25
temporal order with, for example, calls to MPI_BARRIER.
                                                                                      26
                                                                                      27
     Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
                                                                                      28
     order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
                                                                                      29
     received by process 1 using MPI_RECV. (End of advice to users.)
                                                                                      30
                                                                                      31
    Alternatively, a user can impose consistency with nonatomic mode set:
                                                                                      32
                                                                                      33
/* Process 0 */
                                                                                      34
int i, a[10];
                                                                                      35
for ( i=0;i<10;i++)
                                                                                      36
   a[i] = 5;
                                                                                      37
                                                                                      38
MPI_File_open( MPI_COMM_WORLD, "workfile",
                                                                                      39
                MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
                                                                                      40
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                      41
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
                                                                                      42
MPI_File_sync( fh0 ) ;
                                                                                      43
MPI_Barrier( MPI_COMM_WORLD ) ;
                                                                                      44
MPI_File_sync( fh0 ) ;
                                                                                      45
                                                                                      46
/* Process 1 */
                                                                                      47
int b[10];
                                                                                      48
MPI_File_open( MPI_COMM_WORLD, "workfile",
```

1 2 3 4 5 6	<pre>MPI_MODE_RDWR MPI_MODE_CREATE, MPI_INFO_NULL, &fh1); MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL); MPI_File_sync(fh1); MPI_Barrier(MPI_COMM_WORLD); MPI_File_sync(fh1); MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);</pre>
7 8	The "sync-barrier-sync" construct is required because:
9	• The barrier ensures that the write on process 0 occurs before the read on process 1.
10 11 12	• The first sync guarantees that the data written by all processes is transferred to the storage device.
13 14 15	• The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)
16 17	The following program represents an erroneous attempt to achieve consistency by elim- inating the apparently superfluous second "sync" call for each process.
18 19 20 21 22	<pre>/* THIS EXAMPLE IS ERRONEOUS */ /* Process 0 */ int i, a[10] ; for (i=0;i<10;i++) a[i] = 5 ;</pre>
23 24 25 26 27 28 29	<pre>MPI_File_open(MPI_COMM_WORLD, "workfile",</pre>
30 31 32 33 34 35 36 37 38	<pre>/* Process 1 */ int b[10]; MPI_File_open(MPI_COMM_WORLD, "workfile",</pre>
39 40	/* THIS EXAMPLE IS ERRONEOUS */
41 42 43	The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI_FILE_SYNC blocks.
44 45 46 47 48	Advice to users. Some implementations may choose to implement MPI_FILE_SYNC as a temporally synchronizing function. When using such an implementation, the "sync-barrier-sync" construct above can be replaced by a single "sync." The results of using such code with an implementation for which MPI_FILE_SYNC is not temporally synchronizing is undefined. (<i>End of advice to users.</i>)

Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file "myfile." Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

```
int a = 4, b;
                                                                                  39
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                   40
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                  41
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                  42
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
                                                                                  43
MPI_Wait(&reqs[0], &status) ;
                                                                                  44
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
                                                                                   45
MPI_Wait(&regs[1], &status) ;
                                                                                   46
                                                                                   47
```

defines the same ordering as:

 $\frac{24}{25}$

```
1
     int a = 4, b;
\mathbf{2}
     MPI_File_open( MPI_COMM_WORLD, "myfile",
3
                       MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
4
     MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
\mathbf{5}
     MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status ) ;
6
     MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status );
7
     Since
8
9
         • nonconcurrent operations on a single file handle are sequentially consistent, and
10
11
         • the program fragments specify an order for the operations.
12
     MPI guarantees that both program fragments will read the value 4 into b. There is no need
13
     to set atomic mode for this example.
14
          Similar considerations apply to conflicting accesses of the form:
15
16
     MPI_File_write_all_begin(fh,...) ;
17
     MPI_File_iread(fh,...) ;
18
     MPI_Wait(fh,...) ;
19
     MPI_File_write_all_end(fh,...) ;
20
21
         Recall that constraints governing consistency and semantics are not relevant to the
22
     following:
23
     MPI_File_write_all_begin(fh,...) ;
24
     MPI_File_read_all_begin(fh,...) ;
25
     MPI_File_read_all_end(fh,...) ;
26
     MPI_File_write_all_end(fh,...) ;
27
28
     since split collective operations on the same file handle may not overlap (see Section 13.4.5,
```

```
29
30
```

 31

32 33

34

35

36

37 38

39

40

41

42 43

44

13.7 I/O Error Handling

page 548).

By default, communication errors are fatal—MPI_ERRORS_ARE_FATAL is the default error handler associated with MPI_COMM_WORLD. I/O errors are usually less catastrophic (e.g., "file not found") than communication errors, and common practice is to catch these errors and continue executing. For this reason, MPI provides additional error facilities for I/O.

Advice to users. MPI does not specify the state of a computation after an erroneous MPI call has occurred. A high-quality implementation will support the I/O error handling facilities, allowing users to write programs using common practice for I/O. (*End of advice to users.*)

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 8.3, page 356.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in

1 MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is $\mathbf{2}$ MPI_FILE_NULL, 3 I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. 4 The default file error handler has two purposes: when a new file handle is created (by 5MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default 6 $\overline{7}$ error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default 8 file error handler can be changed by specifying MPI_FILE_NULL as the 9 fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error 10 11 handler can be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER. 1213 For communication, the default error handler is inherited from Rationale. 1415

MPI_COMM_WORLD. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI_FILE_NULL. (*End of rationale.*)

13.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

13.9 Examples

13.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function compute_buffer().

```
34
                               _____
                                                                                    35
                                                                                    36
 Function:
                        double_buffer
*
                                                                                    37
                                                                                    38
*
 Synopsis:
                                                                                    39
       void double_buffer(
*
               MPI_File fh,
                                                                                    40
*
                                                              IN
                                                                                    41
*
                MPI_Datatype buftype,
                                                              IN
                                                                                    42
*
                int bufcount
                                                              ΙN
       )
*
                                                                                    43
                                                                                    44
 Description:
                                                                                    45
*
                                                                                    46
*
       Performs the steps to overlap computation with a collective write
                                                                                    47
       by using a double-buffering technique.
*
                                                                                    48
*
```

16

17

18 19 20

21 22

23

 24

25

26 27

28 29

30 31

32

3		
4		
5		
6		
7		
8		
9		
10		
11	MPI_ERR_FILE	Invalid file handle
12	MPI_ERR_NOT_SAME	Collective argument not identical on all
13		processes, or collective routines called in
14		a different order by different processes
15	MPI_ERR_AMODE	Error related to the amode passed to
16		MPI_FILE_OPEN
17	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
18		MPI_FILE_SET_VIEW
19	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
20		a file which supports sequential access only
21	MPI_ERR_NO_SUCH_FILE	File does not exist
22	MPI_ERR_FILE_EXISTS	File exists
23	MPI_ERR_BAD_FILE	
24		Invalid file name (e.g., path name too long) Permission denied
25	MPI_ERR_ACCESS	
26	MPI_ERR_NO_SPACE	Not enough space
27	MPI_ERR_QUOTA	Quota exceeded
28	MPI_ERR_READ_ONLY	Read-only file or file system
	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
29		the file is currently open by some process
30	MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-
31		tered because a data representation identi-
32		fier that was already defined was passed to
33		MPI_REGISTER_DATAREP
34	MPI_ERR_CONVERSION	An error occurred in a user supplied data
35		conversion function.
36	MPI_ERR_IO	Other I/O error
37	Table 19.9	: I/O Error Classes
38	Table 15.5	: 1/O Error Classes
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

```
1
 * Parameters:
                       previously opened MPI file handle
MPI datatype for memory layout
                                                                              2
 *
       fh
                                                                              3
 *
       buftype
                          (Assumes a compatible view has been set on fh)
                                                                              4
 *
                                                                              5
       bufcount
                  # buftype elements to transfer
 *-----*/
                                                                              6
                                                                              7
                                                                              8
/* this macro switches which buffer "x" is pointing to */
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
                                                                              9
                                                                              10
                                                                              11
void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount)
{
                                                                              12
                                                                              13
                            /* status for MPI calls */
                                                                              14
  MPI_Status status;
                                                                              15
  float *buffer1, *buffer2; /* buffers to hold results */
                                                                              16
  float *compute_buf_ptr; /* destination buffer */
                                                                              17
                            /* for computing */
                                                                              18
  float *write_buf_ptr; /* source for writing */
                            /* determines when to quit */
                                                                              19
  int done;
                                                                              20
                                                                              21
  /* buffer initialization */
  buffer1 = (float *)
                                                                              22
                                                                              23
                     malloc(bufcount*sizeof(float)) ;
                                                                              24
  buffer2 = (float *)
                                                                              25
                     malloc(bufcount*sizeof(float)) ;
                                                                              26
   compute_buf_ptr = buffer1 ; /* initially point to buffer1 */
  write_buf_ptr = buffer1 ; /* initially point to buffer1 */
                                                                              27
                                                                              28
                                                                              29
  /* DOUBLE-BUFFER prolog:
                                                                              30
                                                                              31
       compute buffer1; then initiate writing buffer1 to disk
   *
   */
                                                                              32
                                                                              33
   compute_buffer(compute_buf_ptr, bufcount, &done);
                                                                              34
  MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
                                                                              35
  /* DOUBLE-BUFFER steady state:
                                                                              36
                                                                              37
   * Overlap writing old results from buffer pointed to by write_buf_ptr
                                                                              38
   * with computing new results into buffer pointed to by compute_buf_ptr.
                                                                              39
    *
   * There is always one write-buffer and one compute-buffer in use
                                                                              40
                                                                              41
    * during steady state.
                                                                              42
   */
  while (!done) {
                                                                              43
                                                                              44
     TOGGLE_PTR(compute_buf_ptr);
                                                                              45
     compute_buffer(compute_buf_ptr, bufcount, &done);
     MPI_File_write_all_end(fh, write_buf_ptr, &status);
                                                                              46
                                                                              47
     TOGGLE_PTR(write_buf_ptr);
                                                                              48
     MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
```

```
1
         }
\mathbf{2}
3
         /* DOUBLE-BUFFER epilog:
4
           *
                wait for final write to complete.
5
           */
6
         MPI_File_write_all_end(fh, write_buf_ptr, &status);
7
8
9
         /* buffer cleanup */
10
         free(buffer1);
11
         free(buffer2);
12
      }
13
14
      13.9.2 Subarray Filetype Constructor
15
16
17
18
19
20
21
22
23
^{24}
25
26
                                            Process 0
                                                              Process 2
27
                                            Process 1
                                                               Process 3
28
29
                                 Figure 13.4: Example array file layout
30
^{31}
32
33
34
35
36
37
38
39
40
                                            MPI_DOUBLE
                                                          Holes
41
42
                         Figure 13.5: Example local array filetype for process 1
43
44
           Assume we are writing out a 100x100 2D array of double precision floating point num-
      bers that is distributed among 4 processes such that each process has a block of 25 columns
45
46
      (e.g., process 0 has columns 0-24, process 1 has columns 25-49, etc.; see Figure 13.4). To
47
      create the filetypes for each process one could use the following C program (see Section 4.1.3
```

on page 103):

```
1
double subarray[100][25];
                                                                                    \mathbf{2}
MPI_Datatype filetype;
                                                                                    3
int sizes[2], subsizes[2], starts[2];
int rank;
                                                                                    4
                                                                                    5
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                    6
sizes[0]=100; sizes[1]=100;
                                                                                    7
subsizes[0]=100; subsizes[1]=25;
                                                                                    8
starts[0]=0; starts[1]=rank*subsizes[1];
                                                                                    9
                                                                                    10
                                                                                    11
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                           MPI_DOUBLE, &filetype);
                                                                                    12
                                                                                    13
 Or, equivalently in Fortran:
                                                                                    14
                                                                                    15
    double precision subarray(100,25)
                                                                                    16
    integer filetype, rank, ierror
                                                                                    17
    integer sizes(2), subsizes(2), starts(2)
                                                                                    18
                                                                                    19
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
                                                                                   20
    sizes(1)=100
                                                                                   21
    sizes(2)=100
                                                                                   22
    subsizes(1)=100
                                                                                   23
    subsizes(2)=25
                                                                                    ^{24}
    starts(1)=0
                                                                                    25
    starts(2)=rank*subsizes(2)
                                                                                    26
                                                                                   27
    call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
                                                                                   28
                MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                    &
                                                                                   29
                filetype, ierror)
                                                                                   30
                                                                                    ^{31}
                                                                                    32
```

The generated filetype will then describe the portion of the file contained within the process's subarray with holes for the space taken by the other processes. Figure 13.5 shows the filetype created for process 1.

33

	$_{2}^{1}$ ticket266.
	3
	4
	5
	6 7
	8
	9
	10
	11
	12
	13
	14
	16
	17
	18
1	19
	20
	21
	22
	23 24
	25
	26
2	27
	28
	29
	30
	31
	33
	34
	35
з	36
	37
	38
	39 10
	10
	12
	13
	14
	15
	46
	17
4	18

Chapter 14

7 8 9

1516

17

18

19

20

21

22

23 24

25

28

29

31

32

33

ticket266.

ticket266. $\frac{1}{30}$

ticket0.

Tool Support

Introduction 14.1

This chapter discusses a set of interfaces that allows debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 14.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 14.3), which supports the inspection and manipulation of MPI control and performance variables. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

Profiling Interface 14.2

ticket266. 26 [WAS: Chapter] ticket266. 27

Requirements 14.2.1

[WAS: Section]

To meet the requirements for the MPI profiling interface, an implementation of the MPI functions *must*

ticket0. 34 1. provide a mechanism through which all of the MPI defined [functions] functions, exticket0. 35 cept those allowed as macros (See Section 2.6.5)), may be accessed with a name 36 shift. This requires, in C and Fortran, an alternate entry point name, with the prefix ticket229.1. 37 PMPI_ for each MPI function in each provided language binding and language support 38 method. The profiling interface in C++ is described in Section 16.1.10. For routines 39 implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with 40 ticket247-S. 41 a user-defined version.

> For Fortran, the different support methods cause several linker names. Therefore, several profiling routines (with these linker names) are needed for each Fortran MPI routine, as described in Section 16.2.5 on page 651.

2. ensure that those MPI functions that are not replaced may still be linked into an 4647 executable image without causing name clashes.

48

42

43

44

- 1 3. document the implementation of different language bindings of the MPI interface if $\mathbf{2}$ they are layered on top of each other, so that the profiler developer knows whether she ³ ticket0. must implement the profile interface for each binding, or can economise economize 4 by implementing it only for the lowest level routines.
- 4. where the implementation of different language bindings is done through a layered 6 approach ([e.g.]e.g., the Fortran binding is a set of "wrapper" functions that call the 7 ticket0. C implementation), ensure that these wrapper functions are separable from the rest 8 of the library. 9

This separability is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

5. provide a no-op routine MPI_PCONTROL in the MPI library.

14.2.2 Discussion

[WAS: Section]

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which ex-42ecutable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this chapter is only present as justification and discussion of the logic for those requirements.

4647

48

 $\mathbf{5}$

10 11

12

13

1415

16

17

18 19

21

22

23

 24

2526

2728

2930

31

32 33

34

35

36

37

38

39

40 41

43

44

45

ticket266.

 20 ticket 266.

		586			(CHAPTER 14.	TOOL SUPPORT
ticket266.	1 2 3 4		-		vay in which an impl stem (there are dou		
ticket266.	7	[WAS: S			ntation meets the re	quirements abo	ve it is possible for
ticket266.	8 9 10 11 12 13	the impl the user	lementor of • program. erlying MP	the profiling sys She can then co	tem to intercept all ollect whatever infor (through its name s	of the MPI call mation she req	s that are made by uires before calling
+:-l-+966	14	14.2.4	Miscellane	ous Control of P	rofiling		
ticket266.	15 16 17 18	The	ere is a clea	r requirement for	ection to remove sin the user code to be used for (at least) t	able to control	
	19	• En	habling and	disabling profilir	ng depending on the	e state of the cal	culation.
ticket0.	20 21	• Fl	ushing trac	e buffers at non-o	critical points in the	[calculation]cal	culation.
	22 23	• Ac	ding user	events to a trace	file.		
	24 25	The	ese requirer	nents are met by	use of the MPI_PCC	ONTROL.	
	26 27	MPI_PC	ONTROL(I	evel,)			
	28 29	IN	level		Profiling level		
ticket-248T.	30 31	int MPI	_Pcontrol	(const int lev	el,)		
	32 33			el) BIND(C) ENT(IN) :: le	vel		
	34 35 36		NTROL(LEV EGER LEVE				
	37	$\{void M$	IPI::Pcont	rol(const int	level,)(bindir	ng deprecated, se	ee Section 15.2) }
	38 39 40	to the us	ser code. H		no use of this routinn of calls to this routing		-
	41 42 43 44 45 46	to specif vaguenes Hov	fy precisely ss extends wever to pre	the semantics the to the number of ovide some level o	e implementation of at will be provided arguments to the fu f portability of user or certain values of	by calls to MPI inction, and the codes to differen	_PCONTROL. This ir datatypes.
	47 48	• le	vel==0 Pr	ofiling is disabled			

• level==1 Profiling is enabled at a normal default level of detail.

1

39 40

44

47

48

 $_{41}$ ticket 266. $_{42}$ ticket0.

 43 ticket 0.

⁴⁵ ticket0.

⁴⁶ ticket0.

2 • level==2 Profile buffers are [flushed. (This may be a no-op in some profilers).]flushed, ³ ticket0. which may be a no-op in some profilers. 4 5• All other values of level have profile library defined effects and additional arguments. 6 $\overline{7}$ We also request that the default state after MPI_INIT has been called is for profiling to be enabled at the normal default level. (i.e. as if MPI_PCONTROL had just been called 8 9 with the argument 1). This allows users to link with a profiling library and obtain profile output without having to modify their source code at all. 10 ¹¹ ticket0. The provision of MPI_PCONTROL as a no-op in the standard MPI library [allows them to modify their source code to obtain supports the collection of more detailed profiling 12information, but still be able to link exactly the with source same code code that can still ¹³ ticket0. link against the standard MPI library. ¹⁴ ticket0. 15 ticket266. [WAS: Subsection Examples] 16 17 ticket 266. Profiler Implementation [Example 14.2.5 ¹⁸ ticket0. [Suppose that the profiler wishes to] A profiler can accumulate the total amount of data 19 sent by the [MPI_SEND] MPI_SEND function, along with the total elapsed time spent in 20 ticket0. the [function. This could trivially be achieved thus] function, as follows: $_{21}$ ticket 0. 22 static int totalBytes = 0; 23 static double totalTime = 0.0; 24 25int MPI_Send(void* buffer, int count, MPI_Datatype datatype, 26int dest, int tag, MPI_Comm comm) 27{ 28/* Pass on all the arguments */ double tstart = MPI_Wtime(); 29 int extent; 30 int result = PMPI_Send(buffer,count,datatype,dest,tag,comm); 3132 MPI_Type_size(datatype, &extent); /* Compute size */ 33 totalBytes += count*extent; 34 35totalTime += MPI_Wtime() - tstart; /* and time */ 36 37 return result; 38 }

MPI Library Implementation [Example 14.2.6

[On a Unix system, in which the MPI library is implemented in C, then]If the MPI library is implemented in C on a Unix system, then there there are various possible options, of which two of the most obvious are various options, including the two presented here, for supporting [are presented here. Which is better depends on whether the linker and]the name-shift requirement. The choice between these two options [compiler support weak symbols. depends partly on whether the linker and compiler support weak symbols.

```
1
             Systems with Weak Symbols
        2
ticket0. 3
             If the compiler and linker support weak external symbols ([e.g.]e.g., Solaris 2.x, other system
             V.4 machines), then only a single library is required through the use of #pragma weak thus
        4
        5
             #pragma weak MPI_Example = PMPI_Example
        6
        \overline{7}
             int PMPI_Example(/* appropriate args */)
        8
             {
        9
                  /* Useful content */
        10
             }
        11
                  The effect of this #pragma is to define the external symbol MPI_Example as a weak
       12
             definition. This means that the linker will not complain if there is another definition of the
        13
             symbol (for instance in the profiling library), however if no other definition exists, then the
        14
             linker will use the weak definition.
        15
        16
             Systems Without Weak Symbols
        17
        18
             In the absence of weak symbols then one possible solution would be to use the C macro
        19
             pre-processor thus
       20
             #ifdef PROFILELIB
       21
                   ifdef __STDC__
             #
       22
             #
                        define FUNCTION(name) P##name
       23
             #
                   else
       24
             #
                        define FUNCTION(name) P/**/name
       25
             #
                   endif
        26
             #else
       27
                   define FUNCTION(name) name
             #
       28
             #endif
       29
       30
                  Each of the user visible functions in the library would then be declared thus
       ^{31}
             int FUNCTION(MPI_Example)(/* appropriate args */)
       32
             {
        33
                  /* Useful content */
       34
             }
       35
       36
                  The same source file can then be compiled to produce both versions of the library,
       37
             depending on the state of the PROFILELIB macro symbol.
       38
                  It is required that the standard MPI library be built in such a way that the inclusion of
       39
             MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement,
       40
             since it may mean that each external function has to be compiled from a separate file.
```

⁴¹ However this is necessary so that the author of the profiling library need only define those
 ⁴² MPI functions that she wishes to intercept, references to any others being fulfilled by the
 ⁴³ normal MPI library. Therefore the link step can look something like this

45 % cc ... -lmyprof -lpmpi -lmpi

44

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions.
 Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

14.2.7 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions ([e.g.]e.g., a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to "double counting" of the time spent in the inner routine. Since this effect could actually be useful under some circumstances ([e.g., e.g., it might allow one to answer the question "How much time is spent in the point to point routines when they're called from collective functions ?"), we have decided not to enforce any restrictions on the author of the MPI library that would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single threaded world this is easily achieved through use of a static variable in the profiling code that remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded [!]).

Linker Oddities

The Unix linker traditionally operates in one pass : pass: the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard MPI library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the MPI functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be ared out of the base library and into the profiling one.

Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support TYPE(*), DIMENSION(..) choice buffers) imply different linker names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 16.2.5 on page 651.

Multiple Levels of Interception 14.2.8

Unofficial Draft for Comment Only

14

15

18

19

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

38

39 40

41

42

43

44 45

46

47 48

¹⁶ ticket0.

¹⁷ ticket0.

 $_{20}$ ticket0.

589

ticket266.

ticket266.

[WAS: Section] The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

ticket0. 7

ticket206. 12

ticket206. 15

ticket266. 21

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

8

9

10

11

13

14

16

17

18

19

20

22 23

 24

• assuming a particular implementation language[.],

• imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

[Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function.]

[Unfortunately such an implementation may require more cooperation between the different profiling libraries than is required for the single level implementation detailed above.]Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the P^N MPI tool infrastructure [51].

14.3 The MPI Tool Information Interface

25MPI implementations often use internal variables to control their operation and perfor-26mance. Understanding and manipulating these variables can provide a more efficient exe-27cution environment or improve performance for many applications. This section describes 28the MPI tool information interface, which provides a mechanism for MPI implementors to 29expose a set of variables, each of which represents a particular property, setting, or per-30 formance measurement from within the MPI implementation. The interface is split into 31 two parts: the first part provides information about and supports the setting of control 32 variables through which the MPI implementation tunes its configuration. The second part 33 provides access to performance variables that can provide insight into internal performance 34information of the MPI implementation.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control and performance variables exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables that exist in a particular MPI implementation, to query their properties, to retrieve descriptions about their meaning, and to access and, if appropriate, to alter their values.

The MPI tool information interface can be used independently from the MPI com munication functionality. In particular, the routines of this interface can be called before
 MPI_INIT (or equivalent) and after MPI_FINALIZE. In order to support this behavior cleanly,
 the MPI tool information interface uses separate initialization and finalization routines. All
 identifiers used in the MPI tool information interface have the prefix MPI_T_.

⁴⁷ On success, all MPI tool information interface routines return MPI_SUCCESS, otherwise
 ⁴⁸ they return an appropriate and unique return code indicating the reason why the call was

not successfully completed. Details on return codes can be found in Section 14.3.9. However, unsuccessful calls to the MPI tool information interface are not fatal and do not impact the execution of subsequent MPI routines.

Since the MPI tool information interface primarily focuses on tools and support libraries, MPI implementations are only required to provide C bindings for functions introduced in this Section 14.3. Except where otherwise noted, all conventions and principles governing the C bindings of the MPI API also apply to the MPI tool information interface, which is available by including the mpi.h header file. All routines in this interface have local semantics.

Advice to users. The number and type of control variables and performance variables can vary between MPI implementations, platforms and different builds of the same implementation on the same platform as well as between runs. Hence, any application relying on a particular variable will not be portable. Further, there is no guarantee that number of variables, variable indices, and variable names are the same across processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable. (*End of advice to users.*)

14.3.1 Verbosity Levels

The MPI tool information interface provides access to internal configuration and performance information through a set of control and performance variables defined by the MPI implementation. Since some implementations may export a large number of variables, variables are classified by a verbosity level that categorizes both their intended audience (end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. Table 14.1 lists the constants for all possible verbosity levels. The values of the constants are monotonic in the order listed in the table; i.e., MPI_T_VERBOSITY_USER_BASIC < MPI_T_VERBOSITY_USER_DETAIL < ... < MPI_T_VERBOSITY_MPIDEV_ALL.

MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users
MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users
MPI_T_VERBOSITY_USER_ALL	All information of interest to users
MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning
MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
MPI_T_VERBOSITY_TUNER_ALL	All information required for tuning
MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_ALL	All information for MPI implementors
	· ····

Table 14.1: MPI tool information interface verbosity levels.

45 46 47

48

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

17

18

19

20

21

22 23

 24

25

26

27

28

29

30

 31

32

Binding MPI Tool Information Interface Variables to MPI Objects 14.3.2

Each MPI tool information interface variable provides access to a particular control setting 3 or performance property of the MPI implementation. A variable may refer to a specific 4 MPI object such as a communicator, datatype, or one-sided communication window, or the 5variable may refer more generally to the MPI environment of the process. Except for the 6 last case, the variable must be bound to exactly one MPI object before it can be used. Table 14.2 lists all MPI object types to which an MPI tool information interface variable 8 can be bound, together with the matching constant that MPI tool information interface 9 routines return to identify the object type. 10

11	Constant	MPI object
12	MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
13	MPI_T_BIND_MPI_COMM	MPI communicators
14	MPI_T_BIND_MPI_DATATYPE	MPI datatypes
15	MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
16	MPI_T_BIND_MPI_FILE	MPI file handles
17	MPI_T_BIND_MPI_GROUP	MPI groups
18	MPI_T_BIND_MPI_OP	MPI reduction operators
19	MPI_T_BIND_MPI_REQUEST	MPI requests
20	MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
21	MPI_T_BIND_MPI_MESSAGE	MPI message object
22	MPI_T_BIND_MPI_INFO	MPI info object
23		•

 24 25

2627

28

29

30

31

32

33

34

35

Table 14.2: Constants to identify associations of variables.

Some variables have meanings tied to a specific MPI object. Examples Rationale. include the number of send or receive operations using a particular datatype, the number of times a particular error handler has been called, or the communication protocol and "eager limit" used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bounds, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program's execution. (End of rationale.)

36 37 38

39

14.3.3 Convention for Returning Strings

40Several MPI tool information interface functions return one or more strings. These functions 41 have two arguments for each string to be returned: an OUT parameter that identifies a 42pointer to the buffer in which the string will be returned, and an IN/OUT parameter to 43pass the length of the buffer. The user is responsible for the memory allocation of the 44buffer and must pass the size of the buffer (n) as the length argument. Let n be the length 45value specified to the function. On return, the function writes at most n-1 of the string's 46characters into the buffer, followed by a null terminator. If the returned string's length is 47greater than or equal to n, the string will be truncated to n-1 characters. In this case, the 48 length of the string plus one (for the terminating null character) is returned in the length

Unofficial Draft for Comment Only

1

2

argument. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the string plus one in the length argument. If the user passes the null pointer as the length argument, the buffer argument is ignored and nothing is returned.

14.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization routines.

	/ • •	• • • • • • • • • • • • • • • • • • • •
	roduirod	provided 1
	neuuneu.	DIOVIGEUI
MPI_T_INIT_THREAD	(F

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

int MPI_T_init_thread(int required, int *provided)

All programs or tools that use the MPI tool information interface must initialize the MPI tool information interface in the processes that will use the interface before calling any other of its routines. A user can initialize the MPI tool information interface by calling 20MPI_T_INIT_THREAD, which can be called multiple times. In addition, this routine initial-21izes the thread environment for all routines in the MPI tool information interface. Calling 22 this routine when the MPI tool information interface is already initialized has no effect 23beyond increasing the reference count of how often the interface has been initialized. The argument required is used to specify the desired level of thread support. The possible values and their semantics are identical to the ones that can be used with MPI_INIT_THREAD listed in Section 12.4. The call returns in provided information about the actual level of thread support that will be provided by the MPI implementation for calls to MPI tool 28information interface routines. It can be one of the four values listed in Section 12.4.

The MPI specification does not require all MPI processes to exist before the call to 30 MPI_INIT. If the MPI tool information interface is used before MPI_INIT has been called, 31 MPI_T_INIT_THREAD must be called on each process that will use the MPI tool information 32 interface. Processes created by the MPI implementation during MPI_INIT inherit the status 33 of the MPI tool information interface (whether it is initialized or not as well as all active 34 sessions and handles) from the process from which they are created. 35

Processes created at runtime as a result of calls MPI's dynamic process management require their own initialization before they can use the MPI tool information interface.

If MPI_T_INIT_THREAD is called before MPI_INIT_THREAD, Advice to users. the requested and granted thread level for MPI_T_INIT_THREAD may influence the behavior and return value of MPI_INIT_THREAD. The same is true for the reverse order. (End of advice to users.)

Advice to implementors. MPI implementations should strive to make as many control or performance variables available before MPI_INIT (instead of adding them within MPI_INIT) to allow tools the most flexibility. In particular, control variables should be available before MPI_INIT if their value cannot be changed after MPI_INIT. (End of advice to implementors.)

Unofficial Draft for Comment Only

10 11 12

17

18

19

 24

25

26

27

29

36

37 38

39

40

41

4243

44

45

46

47

48

1

 $\mathbf{2}$

3

4

56

7

8

$\frac{1}{2}$	MPI_T_FINALIZE()
3	<pre>int MPI_T_finalize(void)</pre>
4 5 6 7 8 9 10 11 12 13 14 15 16	This routine finalizes the use of the MPI tool information interface and may be called as often as the corresponding MPI_T_INIT_THREAD routine up to the current point of execution. Calling it more times returns a corresponding error code. As long as the number of calls to MPI_T_FINALIZE is smaller than the number of calls to MPI_T_INIT_THREAD up to the current point of execution, the MPI tool information interface remains initialized and calls to its routines are permissible. Further, additional calls to MPI_T_INIT_THREAD after one or more calls to MPI_T_FINALIZE are permissible. Once MPI_T_FINALIZE is called the same number of times as the routine MPI_T_INIT_THREAD up to the current point of execution, the MPI tool information in- terface is no longer initialized. The interface can be reinitialized by subsequent calls to MPI_T_INIT_THREAD. At the end of the program execution, unless MPI_ABORT is called, an application must have called MPI_T_INIT_THREAD and MPI_T_FINALIZE an equal number of times.
17 18	
19 20 21 22 23 24 25 26 27 28	14.3.5 Datatype System All variables managed through the MPI tool information interface represent their values through typed buffers of a given length and type using an MPI datatype (similar to regular send/receive buffers). Since the initialization of the MPI tool information interface is sep- arate from the initialization of MPI, MPI tool information interface routines can be called before MPI_INIT. Consequently, these routines can also use MPI datatypes before MPI_INIT. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before a call to MPI_INIT (or equivalent). MPI_INT
29 30 31 32 33	MPI_UNSIGNED MPI_UNSIGNED_LONG MPI_UNSIGNED_LONG_LONG MPI_COUNT MPI_CHAR
34 35 36 37 38	MPI_DOUBLE Table 14.3: MPI datatypes that can be used by the MPI tool information interface.
 39 40 41 42 	<i>Rationale.</i> The MPI tool information interface relies mainly on unsigned datatypes for integer values since most variables are expected to represent counters or resource sizes. MPI_INT is provided for additional flexibility and is expected to be used mainly for control variables and enumeration types (see below).
43 44 45 46 47 48	Providing all basic datatypes, in particular providing all signed and unsigned variants of integer types, would lead to a larger number of types, which tools need to interpret. This would cause unnecessary complexity in the implementation of tools based on the MPI tool information interface. (<i>End of rationale.</i>)

The MPI tool information interface only relies on a subset of the basic MPI datatypes and does not use any derived MPI datatypes. Table 14.3 lists all MPI datatypes that can be returned by the MPI tool information interface to represent its variables.

Rationale. The MPI tool information interface requires a significantly simpler type system than MPI itself. Therefore, only its required subset must be present before MPI_INIT (or equivalent) and MPI implementations do not need to initialize the complete MPI datatype system. (*End of rationale.*)

For variables of type MPI_INT, an MPI implementation can provide additional information by associating names with a fixed number of values. We refer to this information in the following as an enumeration. In this case, the respective calls that provide additional metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO (Section 14.3.6) and MPI_T_PVAR_GET_INFO (Section 14.3.7), return a handle of type MPI_T_enum that can be passed to the following functions to extract additional information. Thus, the MPI implementation can describe variables with a fixed set of values that each represents a particular state. Each enumeration type can have N different values, with a fixed N that can be queried using MPI_T_ENUM_GET_INFO.

MPI_T_ENUM_GET_INFO(enumtype, num, name, name_len)			
IN	enumtype	enumeration to be queried (handle)	
OUT	num	number of discrete values represented by this enumer- ation (integer)	
OUT	name	buffer to return the string containing the name of the enumeration (string)	
INOUT	name_len	length of the string and/or buffer for $name$ (integer)	

If enumtype is a valid enumeration, this routine returns the number of items represented by this enumeration type. range and the name of the enumeration. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments name and name_len are used to return the name of the enumerations as described in Section 14.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration enumtype can be queried using MPI_T_ENUM_GET_ITEM.

 $\overline{7}$

MPI_T_ENUM_GET_ITEM(enumtype, index, value, name, name_len)

*name, int *name_len)

		· · · · · · · · · · · · · · · · · · ·	
2 3	IN	enumtype	enumeration to be queried (handle)
4	IN	index	number of the value to be queried in this enumeration
5			(integer)
6	OUT	value	variable value (integer)
7	OUT	name	buffer to return the string containing the name of the
8 9			enumeration item (string)
10	INOUT	name_len	length of the string and/or buffer for name (integer)
11			
12	int MPI_T	_enum_get_item(MPI_T_enum	n enumtype, int intex, int value, char

1: 13 14

15

16

17

18

19

2021

22

1

> The arguments name and name_len are used to return the name of the enumeration item as described in Section 14.3.3.

If completed successfully, the routine returns the name/value pair describing the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

14.3.6 Control Variables

23The routines described in this section of the MPI tool information interface specification 24focus on the ability to list, query, and possibly set control variables exposed by the MPI 25implementation. These variables can typically be used by the user to fine tune properties 26and configuration settings of the MPI implementation. On many systems, such variables 27can be set using environment variables, although other configuration mechanisms may be 28available, such as configuration files or central configuration registries. A typical example 29that is available in several existing MPI implementations is the ability to specify an "eager 30 limit", i.e., an upper bound on the size of messages sent or received using an eager protocol.

 31 32

33

Control Variable Query Functions

An MPI implementation exports a set of N control variables through the MPI tool infor-34 mation interface. If N is zero, then the MPI implementation does not export any control 35 variables, otherwise the provided control variables are indexed from 0 to N-1. This index 36 number is used in subsequent calls to identify the individual variables. 37

An MPI implementation is allowed to increase the number of control variables during 38 the execution of an MPI application when new variables become available through dynamic 39 loading. However, MPI implementations are not allowed to change the index of a control 40 variable or delete a variable once it has been added to the set. When variables become 41 inactive, e.g., through dynamic unloading, accessing its value should return a corresponding 42error code. 43

Advice to users. While the MPI tool information interface guarantees that indices or 45 variable properties do not change during a particular run of an MPI program, it does 46 not provide a similar guarantee between runs. (End of advice to users.) 47

48

The following function can be used to query the number of control variables, num_cvar : ¹ ₂					
MPI_T_CV	AR_GET_NUM(num_	.cvar)	3 4		
OUT	num_cvar	returns number of control variables (integer)	5		
int MPI_T	_cvar_get_num(int	*num_cvar)	7		
The fu each variab		$\car{GET_INFO}$ provides access to additional information for	8 9 10 11		
MPI_T_CV	AR_GET_INFO(cvar_ desc_len, bind, s	index, name, name_len, verbosity, datatype, enumtype, desc, cope)	12 13 14		
IN	cvar_index	index of the control variable to be queried, value be- tween 0 and $num_cvar - 1$ (integer)	15 16		
OUT	name	buffer to return the string containing the name of the control variable (string)	17 18		
INOUT	name_len	length of the string and/or buffer for name (integer)	19 20		
OUT	verbosity	verbosity level of this variable (integer)	21		
OUT	datatype	MPI data type of the information stored in the control variable (handle)	22 23		
OUT	enumtype	optional descriptor for enumeration information (han- dle)	24 25 26		
OUT	desc	buffer to return the string containing a description of the control variable (string)	27 28		
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)	29		
OUT	bind	type of MPI object to which this variable must be bound (integer)	30 31 32		
OUT	scope	scope of when changes to this variable are possible (integer)	33 34 35		
<pre>int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len, int</pre>					
calls to thi information The ar as describe If com one. The r	s routine querying in a. An MPI implement guments name and na d in Section 14.3.3. pleted successfully, the	PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same tation is not allowed to alter any of the returned values. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used	39 40 41 42 43 44 45 46 47		

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1).

Unofficial Draft for Comment Only

The argument datatype returns the MPI datatype that is used to represent the control variable.
 If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 14.3.5. If the datatype is not MPI_INT or the argument enumtype is the constant MPI_T_ENUM_NULL, no enumeration type is returned.

The arguments desc and desc_len are used to return a description of the control variable
 as described in Section 14.3.3.

Returning a description is optional. If an MPI implementation decides not to return a
 description, the first character for desc must be set to the null character and desc_len must
 be set to one at the return of this call.

¹³ The parameter bind returns the type of the MPI object to which the variable must be ¹⁴ bound or the value MPI_T_BIND_NO_OBJECT (see Section 14.3.2).

The scope of a variable determines whether changing a variable's value is either local to the process or must be done by the user across multiple processes. The latter is further split into variables that require changes in a group of processes and those that require collective changes among all connected processes. Both cases can require all processes to either be set to consistent (but potentially different) values or to equal values on every participating process. The description provided with the variable must contain an explanation about the requirements and/or restrictions for setting the particular variable.

On successful return from MPI_T_CVAR_GET_INFO, the argument scope will be set to one of the constants listed in Table 14.4.

Scope Constant	Description
MPI_T_SCOPE_READONLY	read-only, cannot be written
MPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation
MPI_T_SCOPE_GROUP	may be writeable, must be done to a group of processes,
	all processes in a group must be set to consistent values
MPI_T_SCOPE_GROUP_EQ	may be writeable, must be done to a group of processes,
	all processes in a group must be set to the same value
MPI_T_SCOPE_ALL	may be writeable, must be done to all processes,
	all connected processes must be set to consistent values
MPI_T_SCOPE_ALL_EQ	may be writeable, must be done to all processes,
	all connected processes must be set to the same value

Table 14.4: Scopes for control variables.

Advice to users. The scope of a variable only indicates if a variable might be changeable; it is not a guarantee that it can be changed at any time. (*End of advice to users.*)

Example: Printing All Control Variables

⁴⁶ Example 14.1

The following example shows how the MPI tool information interface can be used to
 query and print the names of all available control variables.

Unofficial Draft for Comment Only

22

23

38 39

40

41

42 43

```
#include <stdio.h>
#include <stdlibh.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int i, err, num, namelen, bind, verbose, scope;
  int threadsupport;
  char name[100];
  MPI_Datatype datatype;
  err=MPI_T_init_thread(MPI_THREAD_SIGNLE,&threadsupport);
  if (err!=MPI_SUCCESS)
    return err;
  err=MPI_T_cvar_get_num(&num);
  if (err!=MPI_SUCCESS)
    return err;
 for (i=0; i<num; i++) {</pre>
    namelen=100;
    err=MPI_T_cvar_get_info(i, name, &namelen,
            &verbose, &datatype, MPI_T_ENUM_NULL,
            NULL, NULL, /*no description */
            &bind, &scope);
    if (err!=MPI_SUCCESS) return err;
    printf("Var %i: %s\n", i, name);
  }
  err=MPI_T_finalize();
  if (err!=MPI_SUCCESS)
    return 1;
  else
    return 0;
}
```

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user must first allocate a handle of type MPI_T_cvar_handle for the variable by binding it to an MPI object (see also Section 14.3.2).

Rationale. Handles used in the MPI tool information interface are distinct from handles used in the remaining parts of the MPI standard because they must be usable before MPI_INIT and after MPI_FINALIZE. Further, accessing handles, in particular for performance variables, can be time critical and having a separate handle space enables optimizations. (*End of rationale.*)

 $45 \\ 46$

47 48

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10 11

12 13

14 15

16

17

18 19

20 21

22

23

 24

25

26

27 28 29

30

31

32

33

34 35 36

37

38

39 40

41

42

43

1	MPI_T_CV	AR_HANDLE_ALLOC(cvar_ir	ndex, object, handle, count)		
2 3	IN	cvar_index	index of control variable for which handle is to be al- located (index)		
4 5 6	IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)		
7	OUT	handle	allocated handle (handle)		
8 9 10	OUT	count	number of elements used to represent this variable (in- teger)		
11 12 13	int MPI_T	_cvar_handle_alloc(int c MPI_T_cvar_handle *h	<pre>var_index, void *obj_handle, andle, int *count)</pre>		
14 15 16 17 18 19 20	object. The that stores in the argument of the the transmission of transmission of the transmission of	e object is passed in the arg the object's handle. The ha ment handle. Upon successfu	tiable specified by the argument index to an MPI ument obj_handle as an address to a local variable ndle allocated to reference the variable is returned l return, count contains the number of elements (of PI_T_CVAR_GET_INFO call) used to represent this		
21 22 23	Advice to users. The count can be different based on the MPI object to which it was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.				
24 25 26 27 28 29	MPI_ librai addre	COMM_WORLD to this routin ry. Instead, such object han	es to predefined MPI object handles, such as le, since their implementation depends on the MPI idles should be stored in a local variable and the ild be passed into MPI_T_CVAR_HANDLE_ALLOC.		
30 31 32 33 34 35 36	The value of cvar_index should be in the range 0 to $num_cvar - 1$, where num_cvar is the number of available control variables as determined from a prior call to MPI_T_CVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_CVAR_GET_INFO. In the case the bind argument returned by MPI_T_CVAR_GET_INFO equals MPI_T_BIND_NO_OBJECT, the argument obj_handle is ignored.				
37 38	ΜΡΙ Τ Ον	AR_HANDLE_FREE(handle)			
39 40	INOUT	handle	handle to be freed (handle)		
41 42	int MPI_T	_cvar_handle_free(MPI_T_	cvar_handle *handle)		
42 43 44 45 46 47 48	call MPI_T MPI imple	CVAR_HANDLE_FREE to f	a user of the MPI tool information interface should ree the handle and the associated resources in the return, MPI sets the handle to		

Control Va	ariable Access Functions		1 2
MPLT C	VAR_READ(handle, buf		3 4
			5
IN	handle	handle to the control variable to be read (handle)	6
OUT	buf	initial address of storage location for variable value (choice)	7 8
			9
int MPI_	<pre>[] [] [] [] [] [] [] [] [] [] [] [] [] [</pre>	var_handle handle, void* buf)	10
			11
		e of the control variable identified by the argument handle identified by the parameter buf . The user must ensure that	12
		we to hold the entire value of the control variable (based on	13
		rom prior corresponding calls to MPI_T_CVAR_GET_INFO	14 15
	T_CVAR_HANDLE_ALI		15
		,,	17
		C)	18
MPI_I_C	VAR_WRITE(handle, bu		19
IN	handle	handle to the control variable to be written (handle)	20
IN	buf	initial address of storage location for variable value	21
		(choice)	22
			23
int MPI_	<pre>I_cvar_write(MPI_T_c</pre>	<pre>cvar_handle handle, const void* buf)</pre>	24 25
This	routine sets the value o	f the control variable identified by the argument handle to	25
		tified by the parameter buf . The user must ensure that the	27
	buffer is of the appropriate size to hold the entire value of the control variable (based on the		
		m prior corresponding calls to MPI_T_CVAR_GET_INFO	29
	T_CVAR_HANDLE_ALI		30
	_	scope (as returned by a prior corresponding	31
		ny write call to this variable must be issued by the user 10.5 (d) MDI and 10.5 (d)	32
	N Contraction of the second se	ection 10.5.4) MPI processes. If the variable has a group iable must be issued by the user in all MPI processes in	33
		bed by the MPI implementation in the description by the	34 35
	VAR_GET_INFO.		36
In bo	th cases, the user must	ensure that the writes in all processes are consistent. If	37
the scope	is either $MPI_T_SCOPE_$	GLOBAL_EQ or MPI_T_SCOPE_GROUP_EQ this means that	38
	-	be set to the same value.	39
		ge the variable at the time the call is made, the function	40
		SETNOTNOW, if there may be a later time at which the	41
		RR_CVAR_SETNEVER, if the variable cannot be set for the coution	42
remainder	of the application's ex		43
Example:	Reading the Value of a (Control Variable	44 45
Lample.	incounts the value of a v		45
Erroreel	14.9		47
Example	14.2		48

The following example shows a routine that can be used to query the value with a control variable with a given index. The example assumes that the variable is intended to be bound to an MPI communicator.

```
int getValue_int_comm(int index, MPI_Comm comm, int *val) {
\mathbf{5}
6
              int err,count;
             MPI_T_cvar_handle handle;
7
8
              /* This is example assumes that the variable index */
9
              /* can be bound to a communicator */
10
11
              err=MPI_T_cvar_handle_alloc(index,&comm,&handle,&count);
12
              if (err!=MPI_SUCCESS) return err;
13
14
              /* The following assumes that the variable is */
15
              /* represented by a single integer */
16
17
18
              err=MPI_T_cvar_read(handle,val);
19
              if (err!=MPI_SUCCESS) return err;
20
              err=MPI_T_cvar_handle_free(&handle);
21
22
             return err;
     }
23
^{24}
```

²⁵ 14.3.7 Performance Variables

The following section focuses on the ability to list and query performance variables provided by the MPI implementation. Performance variables provide insight into MPI implementation specific internals and can represent information such as the state of the MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data for submodules, or queue sizes and lengths.

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics,
 possible datatypes, basic behavior, its starting value, whether it can overflow, and when
 and how an MPI implementation can change the variable's value. The starting value is the
 value the variable assumes when it is used for the first time or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is
 up to the user to appropriately protect against this, e.g., by frequently reading and
 reseting the variable value. (End of advice to users.)

48

43

26

32

33

34

35

36 37

38

1

 $\mathbf{2}$

3

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (*End* of advice to implementors.)

The classes are defined by the following constants:

• MPI_T_PVAR_CLASS_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 14.3.5. The starting value is the current state of the implementation at the time the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_LEVEL

A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_SIZE

A performance variable in this class represents a value that is the fixed size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_PERCENTAGE

The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an MPI_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_HIGHWATERMARK

A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is non-negative and grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30 31

32

33

34

35

36

37

38

39

40 41

42

43 44

 $45 \\ 46$

47

1 MPI_T_PVAR_CLASS_LOWWATERMARK 2 A performance variable in this class represents a value that describes the low wa-3 termark utilization of a resource. The value of a variable of this class is non-4 negative and decreases monotonically from the initialization or reset of the vari-5able. It can be represented by one of the following datatypes: MPI_UNSIGNED, 6 MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The 7 starting value is the current utilization level of the resource at the time the start-8 ing value is set. MPI implementations must ensure that variables of this class cannot 9 overflow. 10 MPI_T_PVAR_CLASS_COUNTER 11 A performance variable in this class counts the number of occurrences of a specific 12event (e.g., the number of memory allocations within an MPI library). The value of 13 a variable of this class increases monotonically from the initialization or reset of the 14performance variable by one for each specific event that is observed. Values must be 15non-negative and represented by one of the following datatypes: MPI_UNSIGNED, 16MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG. The starting value for 17 variables of this class is 0. Variables of this class can overflow. 18 19 MPI_T_PVAR_CLASS_AGGREGATE 20The value of a performance variable in this class is an an aggregated value that repre-21sents a sum of arguments processed during a specific event (e.g., the amount of mem-22 ory allocated by all memory allocations). This class is similar to the counter class, 23but instead of counting individual events, the value can be incremented by arbitrary 24 amounts. The value of a variable of this class increases monotonically from the initial-25ization or reset of the performance variable. It must be non-negative and represented 26by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, 27MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value for variables 28of this class is 0. Variables of this class can overflow. 29 30 MPI_T_PVAR_CLASS_TIMER 31The value of a performance variable in this class represents the aggregated time 32 that the MPI implementation spends executing a particular event, type of event, 33 or section of the MPI library. This class has the same basic semantics as 34MPI_T_PVAR_CLASS_AGGREGATE, but explicitly records a timing value. The 35 value of a variable of this class increases monotonically from the initialization 36 or reset of the performance variable. It must be non-negative and represented 37 by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, 38

MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value for variables of this class is 0. If the type MPI_DOUBLE is used, the units representing time in this datatype must match the units used by MPI_WTIME. Otherwise, the time units should be documented, e.g., in the description returned by MPI_T_PVAR_GET_INFO. Variables of this class can overflow.

• MPI_T_PVAR_CLASS_GENERIC

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable specific and implementation defined.

47 48

39

40

41

42

43

44

45

Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any performance variables, otherwise the provided performance variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or delete a variable once it has been added to the set. When variables become inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, N:

MPI_T_PVAR_GET_NUM(num_pvar)

OUTnum_pvarreturns number of performance variables (integer)

int MPI_T_pvar_get_num(int *num_pvar)

The function MPI_T_PVAR_GET_INFO provides access to additional information for each variable.

INI		len, bind, readonly, continuous, atomic)
IN	pvar_index	index of the performance variable to be queried b tween 0 and $num_pvar - 1$ (integer)
OUT	name	buffer to return the string containing the name of the performance variable (string)
INOUT	name_len	length of the string and/or buffer for name (integer
OUT	verbosity	verbosity level of this variable (integer)
OUT	var_class	class of performance variable (integer)
OUT	datatype	MPI datatype of the information stored in the performance variable (handle)
OUT	enumtype	optional descriptor for enumeration information (ha dle)
OUT	desc	buffer to return the string containing a description the performance variable (string)
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)
OUT	bind	type of MPI object to which this variable must bound (integer)
OUT	readonly	flag indicating whether the variable can be written/re $(integer)$
OUT	continuous	flag indicating whether the variable can be started an stopped or is continuously active (integer)
OUT	atomic	flag indicating whether the variable can be atomical read and reset (integer)
int MPI_	<pre>*verbosity, ir *enumtype, cha</pre>	<pre>pvar_index, char *name, int *name_len, int nt *var_class, MPI_Datatype *datatype, MPI_T_enu ar *desc, int *desc_len, int *bind, int t *continuous, int *atomic)</pre>
calls to the information	nis routine querying in on. An MPI implement	I_T_PVAR_GET_INFO for a particular variable, subsequent formation about the same variable must return the same variable is not allowed to alter any of the returned values.
The s	-	name_len are used to return the name of the performance
		14.3.3. If completed successfully, the routine is require th one.
variable a	a name of at least leng	· · · ·
variable a to return	a name of at least leng argument verbosity retu	μ irns the verbosity level of the variable (see Section 14.3.1
variable a to return The a The c	argument verbosity returns of the performanc	rns the verbosity level of the variable (see Section 14.3.1 e variable is returned in the parameter var_class. The cla
variable a to return The a The o must be o	argument verbosity retu class of the performanc ne of the constants def	e variable is returned in the parameter var_class . The cla fined in Section 14.3.7.
variable a to return The a The o must be o The o	argument verbosity retuined and the performance of the constants defined and the constants defined and the name of	e variable is returned in the parameter var_class. The cla fined in Section 14.3.7. he and the class of the performance variable must be uniq
variable a to return The a The c must be o The c with response	argument verbosity retu class of the performanc ne of the constants def combination of the name ect to all other names f	e variable is returned in the parameter var_class. The cla

variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (End of advice to implementors.)

The argument datatype returns the MPI datatype that is used to represent the performance variable.

If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used as described in Section 14.3.5 to gather more information. If the datatype is not MPI_INT or the argument enumtype is the constant MPI_T_ENUM_NULL, no emumeration type is returned.

Returning a description is optional. If an MPI implementation decides not to return a description, the first character for desc must be set to the null character and desc_len must be set to one at the return from this function.

The parameter bind returns the type of the MPI object to which the variable must be bound or the value MPI_T_BIND_NO_OBJECT (see Section 14.3.2).

Upon return, the argument readonly is set to zero if the variable can be written or reset by the user. It is set to one if the variable can only be read.

Upon return, the argument continuous is set to zero if the variable can be started and stopped by the user, i.e., it is possible for the user to control if and when the value of a variable is updated. It is set to one if the variable is always active and cannot be controlled 20by the user.

Upon return, the argument **atomic** is set to zero if the variable cannot be atomically read and reset. Only variables for which the call sets **atomic** to one, can be used in a call to MPI_T_PVAR_READRESET.

Performance Experiment Sessions

Within a single program, multiple components can use the MPI tool information interface. To avoid collisions with respect to accesses to performance variables, users of the MPI tool information interface must first create a session. Subsequent calls accessing performance variables can then be made within the context of this session. Any call executed in a session must not influence the results in any other session.

	AR_SESSION_CREATE(session	n)	34	
	, , , , , , , , , , , , , , , , , , ,	,		
OUT	session	identifier of performance session (handle)	36	
			37	
int MPI_T	_pvar_session_create(MPI_	T_pvar_session *session)	38	
This call creates a new session for accessing performance variables and returns a handle				
	for this session in the argument session of type MPI_T_pvar_session.			
			42	
MPI_T_PV	AR_SESSION_FREE(session)		43	
INOUT	session	identifier of performance experiment session (handle)	44	
	Session		45	
			46	
int MPI_T	nt MPI_T_pvar_session_free(MPI_T_pvar_session *session) 47			

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16

17

18

19

21

22

23

242526

27

28

29

30

 31

32 33

1 2 3 4	longer be	<u> </u>	sion. Calls to the MPI tool information interface can no of a session after it is freed. On a successful return, MPI T_PVAR_SESSION_NULL.			
5	Handle Al	Handle Allocation and Deallocation				
6 7 8 9		Before using a performance variable, a user must first allocate a handle of type MPI_T_pvar_handle for the variable by binding it to an MPI object (see also Section 14.3.2).				
10 11	MPI_T_P	VAR_HANDLE_ALLOC(s	ession, pvar_index, obj_handle, handle, count)			
11	IN	session	identifier of performance experiment session (handle)			
13 14	IN	pvar_index	index of performance variable for which handle is to be allocated (integer)			
15 16 17	IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)			
18	OUT	handle	allocated handle (handle)			
19 20 21	OUT	count	number of elements used to represent this variable (in-teger)			
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	This MPI obje the argun The hand successful previous I Adv was mat It is MPI libra add (En	void *obj_handl routine binds the perfor- ct in the session identifi- nent obj_handle as an ad lle allocated to reference return, count contains to MPI_T_PVAR_GET_INFO vice to users. The coun- bound. For example, var- sches the size of the comm s not portable to pass ref _COMM_WORLD, to this ary. Instead, such object ress of this local variables d of advice to users.)	ferences to predefined MPI object handles, such as routine, since their implementation depends on the MPI at handles should be stored in a local variable and the s should be passed into MPI_T_PVAR_HANDLE_ALLOC.			
41 42 43 44 45 46 47 48	The value of index should be in the range 0 to num_pvar - 1, where num_pvar is the number of available control variables as determined from a prior call to MPI_T_PVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_PVAR_GET_INFO. In the case the bind argument equals MPI_T_BIND_NO_OBJECT, the argument obj_handle is ignored.					

MPI_T_F	VAR_HANDLE_FR	EE(session, handle)
IN	session	identifier of performance experiment session (handle)
INOUT	handle	handle to be freed (handle)
int MPI_	T_pvar_handle_fi *handle)	ree(MPI_T_pvar_session session, MPI_T_pvar_handle
call MPI_ rameter s	T_PVAR_HANDLE ession and the asso	ager needed, a user of the MPI tool information interface should E_FREE to free the handle in the session identified by the pa- ociated resources in the MPI implementation. On a successful to MPI_T_PVAR_HANDLE_NULL.
Starting a	nd Stopping of Perf	ormance Variables
continuou any time stopped s	sly operating once but they cannot b	have the continuous flag set during the query operation are a handle has been allocated. Such variables may be queried at be started or stopped by the user. All other variables are in a adde has been allocated; their values are not updated until they er.
MPI_T_P	VAR_START(sessio	on, handle)
IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)
int MPI_	T_pvar_start(MP]	I_T_pvar_session session, MPI_T_pvar_handle handle)
rameter h If the attempts which ha variables Continuo	andle in the session e constant MPI_T_P to start all variab ndles have been al are started success	he performance variable with the handle identified by the pa- n identified by the parameter session. VAR_ALL_HANDLES is passed in handle, the MPI implementation les within the session identified by the parameter session for located. In this case, the routine returns MPI_SUCCESS if all offully, otherwise MPI_T_ERR_PVAR_NOSTARTSTOP is returned. ariables that are already started are ignored when is specified.
MPI_T_F	VAR_STOP(session	, handle)
IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)
int MPI_	.T_pvar_stop(MPI_	_T_pvar_session session, MPI_T_pvar_handle handle)
This eter hand If the	functions stops the le in the session ide e constant MPI_T_P	e performance variable with the handle identified by the param- entified by the parameter session. VAR_ALL_HANDLES is passed in handle, the MPI implementation les within the session identified by the parameter session for

1	which har	dles have been allocated. In	this case, the routine returns MPI_SUCCESS if all	
2	variables a	are stopped successfully, otherw	wise MPI_T_ERR_PVAR_NOSTARTSTOP is returned.	
3	Continuou	is variables and variables that	t are already stopped are ignored when	
4	MPI_T_PV	$\ensuremath{AR_ALL_HANDLES}$ is specified.		
5				
6	Performan	ce Variable Access Functions		
7				
8				
9 10	MPI_T_P	VAR_READ(session, handle, but	f)	
11	IN	session	identifier of performance experiment session (handle)	
12	IN	handle	handle of a performance variable (handle)	
13	OUT	buf	initial address of storage location for variable value	
14	••••		(choice)	
15				
16	int MPI '	I pvar read(MPI T pvar sea	ssion session, MPI_T_pvar_handle handle,	
17 18	_	void* buf)		
19	T L - 1		ing the surface of the menformer of surface is here with the	
20			ies the value of the performance variable with the y the parameter session and stores the result in the	
21			The user is responsible to ensure that the buffer	
22		· ·	ntire value of the performance variable (based on	
23			ne corresponding previous calls to	
24	•	· · · · · · · · · · · · · · · · · · ·	VAR_HANDLE_ALLOC, respectively).	
25			NDLES cannot be used as an argument for the func-	
26		T_PVAR_READ.		
27				
28		VAR_WRITE(session,handle, bu	.f)	
29 30		× ×	,	
31	IN	session	identifier of performance experiment session (handle)	
32	IN	handle	handle of a performance variable (handle)	
33	IN	buf	initial address of storage location for variable value	
34			(choice)	
35				
36	int MPI_	I_pvar_write(MPI_T_pvar_se	ession session, MPI_T_pvar_handle handle,	
37		<pre>const void* buf)</pre>		
38	The N	API T PVAR WRITE call atte	mpts to write the value of the performance variable	
39			ter handle in the session identified by the parameter	
40			d in the buffer identified by the parameter buf . The	
41 42			appropriate size to hold the entire value of the per-	
43	formance variable (based on the datatype and count returned by the corresponding previous			
44	calls to MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively).			
45			e variable, the function returns	
46	MPI_T_ERR_PVAR_NOWRITE.			
47			NDLES cannot be used as an argument for the func-	
48	tion MPI_	T_PVAR_WRITE.		

CHAPTER 14. TOOL SUPPORT

MPI_	T_PVAR_RESET(session	ı, handle)	1	
IN	session	identifier of performance experiment session (handle)	2 3	
IN	handle		4	
			5	
int 1	MPI_T_pvar_reset(MPI		6	
r	The MPI_T_PVAR_RESE	T call sets the performance variable with the handle identified	7 8	
by th	e parameter handle to its	s starting value specified in Section 14.3.7. If it is not possible	8 9	
		nction returns MPI_T_ERR_PVAR_NOWRITE.	10	
			11	
	•	es within the session identified by the parameter session for	12	
		bocated. In this case, the routine returns MPI_SUCCESS if all ly, otherwise MPI_T_ERR_PVAR_NOWRITE is returned. Read-	13	
		en MPI_T_PVAR_ALL_HANDLES is specified.	14	
omj			15	
			16 17	
MPI_	T_PVAR_READRESET(session, nandle, but)	18	
IN	session	identifier of performance experiment session (handle)	19	
IN	handle	handle of a performance variable (handle)	20	
OU	T buf	initial address of storage location for variable value	21	
		(choice)	22	
			23	
int l	•	(MPI_T_pvar_session session, MPI_T_pvar_handle	24 25	
	handle, voi	d* buf)	26	
5	This call atomically com	bines the functionality of $MPI_T_PVAR_READ$ and	27	
MPI_	T_PVAR_RESET with t	he same semantics as if these two calls were called separately.	28	
		variable are not supported, this routine returns	29	
	ERR_NOATOMIC.		30	
	ion MPI_T_PVAR_REA		31 32	
Tunet			33	
	Advice to implementors	3. Sampling based tools rely on the ability to call the MPI	34	
	tool information interfa	ce, in particular routines to start, stop, read, write and reset	35	
	- · · · · · · · · · · · · · · · · · · ·		36	
	_		37	
			38	
			$\frac{39}{40}$	
	should clearly document any restrictions on the program contexts in which the MPI			
			$41 \\ 42$	
	outside of all signals or outside a specific set of signals. Any restrictions could be docu-			
mented, for example, through the description returned by $MPI_T_PVAR_GET_INFO.$				
	(End of advice to implementors.)			
	Rationale. All routines to read, write or reset performance variables require the			
		keeps the interface consistent and allows the use	47 48	

1

 $\mathbf{2}$

3 4

5 6

7

8

9

10

11

12

18

MPI_T_PVAR_ALL_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

Example: Tool to Detect Receives with Long Unexpected Message Queues

Example 14.3

The following example shows a sample tool to identify receive operations that occur during times with long message queues. This examples assumes that the MPI implementation exports a variable with the name "MPI_T_UMQ_LENGTH" to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to MPI_INIT), (2) the test for long unexpected message queues (by intercepting calls to MPI_RECV), and (3) the clean up phase (by intercepting the call to MPI_FINALIZE. To capture all receives, the example would have to be extended to have similar wrappers for all receive operations.

Part 1— Initialization: During initialization, the tool searches for the variable and, once
 the right index is found, allocates a session and a handle for the variable with the found
 index, and starts the performance variable.

```
22
     #include <stdio.h>
23
     #include <stdlib.h>
24
     #include <assert.h>
25
     #include <mpi.h>
26
27
     /* Global variables for the tool */
28
     static MPI_T_pvar_session session;
29
     static MPI_T_pvar_handle handle;
30
^{31}
     int MPI_Init(int *argc, char ***argv) {
32
              int err, num, i, index, namelen, verbosity;
33
                   int var_class, bind, threadsup;
34
              int readonly, continuous, atomic, count;
35
              char name[17];
36
             MPI_Comm comm;
37
             MPI_Datatype datatype;
38
             MPI_T_enum enumtype;
39
40
              err=PMPI_Init(argc,argv);
41
              if (err!=MPI_SUCCESS) return err;
42
43
              err=PMPI_T_init_thread(MPI_THREAD_SINGLE,&threadsup);
44
              if (err!=MPI_SUCCESS) return err;
45
46
              err=PMPI_T_pvar_get_num(&num);
47
              if (err!=MPI_SUCCESS) return err;
48
```

}

{

```
1
        index=-1;
                                                                                      \mathbf{2}
        i=0;
                                                                                      3
        while ((i<num) && (index<0)) {
                 namelen=17;
                                                                                      4
                 err=PMPI_T_pvar_get_info(i, name, namelen, &verbosity,
                                                                                      5
                                                                                      6
                         &var_class, &datatype, &enumtype, &bind,
                                                                                      7
                         &readonly, &continuous, &atomic);
                                                                                      8
                 if (strcmp(name, "MPI_T_UMQ_LENGTH")==0) index=i;
                                                                                      9
                 i++; }
                                                                                      10
                                                                                     11
        /* this could be handled in a more flexible way for a generic tool */
        assert(index>=0);
                                                                                     12
        assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
                                                                                     13
                                                                                     14
        assert(datatype==MPI_INT);
                                                                                     15
        assert(bind==MPI_T_BIND_MPI_COMM);
                                                                                     16
                                                                                      17
        /* Create a session */
                                                                                     18
        err=PMPI_T_pvar_session_create(&session);
                                                                                     19
        if (err!=MPI_SUCCESS) return err;
                                                                                     20
                                                                                     21
        /* Get a handle and bind to MPI_COMM_WORLD */
        comm=MPI_COMM_WORLD;
                                                                                     22
                                                                                     23
        err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
                                                                                     24
        if (err!=MPI_SUCCESS) return err;
                                                                                     25
                                                                                     26
        /* this could be handled in a more flexible way for a generic tool */
        assert(count==1);
                                                                                     27
                                                                                     28
                                                                                     29
        /* Start variable */
                                                                                     30
        err=PMPI_T_pvar_start(session, handle);
                                                                                     31
        if (err!=MPI_SUCCESS) return err;
                                                                                     32
                                                                                     33
        return MPI_SUCCESS;
                                                                                     34
                                                                                     35
                                                                                     36
Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the
                                                                                     37
tool reads the unexpected queue length through the matching performance variable and
                                                                                     38
compares it against a predefined threshold.
                                                                                     39
#define THRESHOLD 5
                                                                                      40
                                                                                     41
                                                                                     42
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
                          MPI_Comm comm, MPI_Status *status)
                                                                                     43
                                                                                     44
                                                                                     45
        int value, err;
                                                                                     46
                                                                                      47
        if (comm==MPI_COMM_WORLD) {
                                                                                     48
                 err=PMPI_T_pvar_read(session, handle, &value);
```

```
1
                       if ((err==MPI_SUCCESS) && (value>THRESHOLD))
2
                       {
3
                                   /* tool identified receive called with long UMQ */
4
                                /* execute tool functionality, */
5
                                /* e.g., gather and print call stack */
6
                       }
7
              }
8
9
              return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
10
     }
11
12
     Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information inter-
13
     face is finalized.
14
15
     int MPI_Finalize()
16
     {
17
              int err;
18
              err=PMPI_T_handle_free(&session, &handle);
19
              err=PMPI_T_session_free(&session);
20
              err=PMPI_T_finalize();
21
              return PMPI_Finalize();
22
     }
23
^{24}
     14.3.8 Variable Categorization
```

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. For example, an MPI implementation could group all control and performance variables that refer to message transfers in the MPI implementation and thereby distinguish them from variables that refer to local resources such as memory allocations or other interactions with the operating system.

Categories can also contain other categories to form a hierarchical grouping. Categories can never include themselves, either directly or transitively within other included categories. Expanding on the example above, this allows MPI to refine the grouping of variables referring to message transfers into variables to control and monitor message queues, message matching activities and communication protocols. Each of these groups of variables would be represented by a separate category and these categories would then be listed in a single category representing variables for message transfers.

The category information may be queried in a fashion similar to the mechanism for querying variable information. The MPI implementation exports a set of N categories via the MPI tool information interface. If N = 0, then the MPI implementation does not export any categories, otherwise the provided categories are indexed from 0 to N - 1. This index number is used in subsequent calls to functions of the MPI tool information interface to identify the individual categories.

An MPI implementation is permitted to increase the number of categories during the execution of an MPI program when new categories become available through dynamic loading. However, MPI implementations are not allowed to change the index of a category or delete it once it has been added to the set.

25

	· · · · ·	e allowed to add variables to categories, but they n categories or change the order in which they are	1 2 3	
The fo	llowing function can be used	to query the number of control variables, N .	4 5	
MPLT CA	TEGORY_GET_NUM(num_ca	t)	6 7	
OUT	•	,	8	
001	num_cat	current number of categories (integer)	9	
int MPT T	_category_get_num(int *nu	um cat)	10	
			11	
Individ	dual category information can	then be queried by calling the following function:	12 13	
MPI_T_CA	TEGORY_GET_INFO(cat_ind num_categories)	ex, name, name_len, desc, desc_len, num_cvars, num_pv		
IN	cat_index	index of the category to be queried (integer)	16 17	
OUT	name	buffer to return the string containing the name of the	18	
001	lidilic	category (string)	19	
INOUT	name_len	length of the string and/or buffer for name (integer)	20	
OUT	desc		21	
001	desc	buffer to return the string containing the description of the category (string)	22 23	
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)	24	
OUT	num_cvars	number of control variables in the category (integer)	25 26	
OUT	num_pvars	number of performance variables in the category (in-	20	
		teger)	28	
OUT	num_categories	number of categories contained in the category (inte-	29	
		ger)	30 31	
			31	
int MPI_T		at_index, char *name, int *name_len, char	33	
		, int *num_cvars, int *num_pvars, int	34	
	<pre>*num_categories)</pre>		35	
		are used to return the name of the category as	36	
	n Section 14.3.3.	where the set has the set of the second second by	37	
	-	name of at least length one. This name must be or categories used by the MPI implementation.	38 39	
		e used to return the description of the category as	40	
	n Section 14.3.3.	a about to retain the description of the category as	41	
		If an MPI implementation decides not to return a	42	
description	, the first character for $desc$ n	nust be set to the null character and $desc_len$ must	43	
	ne at the return of this call.		44	
		control variables, performance variables and other	45	
-	categories contained in the queried category in the arguments num_cvars, num_pvars, and 4um_categories, respectively. 4			
num_catego	ones, respectively.		47 48	

```
1
      MPI_T_CATEGORY_GET_CVARS(cat_index, len, indices)
2
        IN
                  cat_index
                                                index of the category to be queried, in the range [0, N-
3
                                                1] (integer)
4
        IN
                  len
                                                the length of the indices array (integer)
5
6
        OUT
                  indices
                                                an integer array of size len, indicating control variable
7
                                                indices (array of integers)
8
9
      int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
10
          MPI_T_CATEGORY_GET_CVARS can be used to query which control variables are
11
      contained in a particular category. A category contains zero or more control variables.
12
13
14
      MPI_T_CATEGORY_GET_PVARS(cat_index,len,indices)
15
        IN
                  cat_index
                                                index of the category to be queried, in the range [0, N-
16
17
                                                1] (integer)
18
        IN
                  len
                                                the length of the indices array (integer)
19
        OUT
                  indices
                                                an integer array of size len, indicating performance
20
                                                variable indices (array of integers)
21
22
      int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
23
^{24}
          MPI_T_CATEGORY_GET_PVARS can be used to query which performance variables
25
      are contained in a particular category. A category contains zero or more performance
26
      variables.
27
28
      MPI_T_CATEGORY_GET_CATEGORIES(cat_index,len,indices)
29
30
        IN
                  cat_index
                                                index of the category to be queried, in the range [0, N-
^{31}
                                                1] (integer)
32
        IN
                  len
                                                the length of the indices array (integer)
33
34
        OUT
                  indices
                                                an integer array of size len, indicating category indices
35
                                                (array of integers)
36
37
      int MPI_T_category_get_categories(int cat_index, int len, int indices[])
38
          MPI_T_CATEGORY_GET_CATEGORIES can be used to query which other categories
39
      are contained in a particular category. A category contains zero or more other categories.
40
          As mentioned above, MPI implementations can grow the number of categories as well
41
      as the number of variables or other categories within a category. In order to allow users
42
      of the MPI tool information interface to quickly check whether new categories have been
43
      added or new variables or categories have been added to a category, MPI maintains a
44
      virtual timestamp. This timestamp is monotonically increasing during the execution and is
45
      returned by the following function:
46
47
48
```

MPI_T_CA	TEGORY_CHANGED(stamp)		1
OUT	stamp	a virtual time stamp to indicate the last change to the	2 3
		categories (integer)	4
int MDT T	astanony abannod (int wat		5
	_category_changed(int *st	-	6
	—	e return the same timestamp, it is guaranteed that	7 8
0	, S	between the two calls. If the timestamp retrieved e categories have been added or expanded.	9
	scond can is inglicit, then some	categories have been added of enpanded.	10
		value is purely virtual and only intended to check	11
	nanges in the category information of advice to users.)	tion. It should not be used for any other purpose.	12 13
(Lina	of untice to users.)		14
		s by MPI_T_CATEGORY_GET_CVARS,	15
		PI_T_CATEGORY_GET_CATEGORIES can be used	16
-	TEGORY_GET_INFO, respect	MPI_T_PVAR_GET_INFO and ively	17 18
	· · · · · · · · · · · · · · · · · · ·	g the arrays passed into the functions	19
MPI_T_CA	TEGORY_GET_CVARS, MPI_	T_CATEGORY_GET_PVARS and	20
		. Starting from array index 0, each function writes	21
-	•	ne category contains more than len elements, the size len. Otherwise, the entire set of elements is	22 23
	•	e array, and any remaining array entries are not	24
modified.			25
			26
14.3.9 Re	eturn Codes for the MPI tool	information interface	27 28
		tool information interface return an integer return	29
· · · · · · · · · · · · · · · · · · ·		the function has completed successfully or aborted	30
		n code indicates the reason for not completing the ed by an routine impact the execution of the MPI	31
		dlers. The execution of the MPI process continues	32 33
	-	wever, the MPI implementation is not required to	34
		ser passes invalid parameter values to any routine	35
	or of the implementation is un turn codes with the prefix MI	PI_T_ must be unique values and cannot overlap	36
	ther return values returned by		37 38
-			39
14.3.10 F	Profiling Interface		40
All require	ements for the profiling interf	aces, as described in Section 14.2, also apply to	41
		rules, guidelines, and recommendations from Sec-	42 43
tion 14.2 a	pply equally to calls defined a	s part of the MPI tool information interface.	43 44
			45
			46
			47
			48

Return Code	Description
Return Codes for all Functions in t	the MPI tool information interface
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOTINITIALIZED	Interface not initialized
MPI_T_ERR_CANTINIT	Interface not in the state to be initialized
Return Codes for Datatype Function	ons: MPI_T_ENUM_*
MPI_T_ERR_INVALIDINDEX	The enumeration index is invalid or has been de
MPI_T_ERR_INVALIDITEM	The item index queried is out of range
	(for MPI_T_ENUMITEM only)
Return Codes for variable and cate	egory query functions: MPI_T_*_GET_INFO
MPI_T_ERR_INVALIDINDEX	The variable or category index is invalid
Return Codes for Handle Function	s: MPI_T_*_ALLOCATE,FREE
MPI_T_ERR_INVALIDINDEX	The variable index is invalid or has been deleted
MPI_T_ERR_INVALIDHANDLE	The handle is invalid
MPI_T_ERR_OUTOFHANDLES	No more handles available
Return Codes for Session Function	s: MPI_T_PVAR_SESSION_*
MPI_T_ERR_OUTOFSESSIONS	No more sessions available
MPI_T_ERR_INVALIDSESSION	Session argument is not a valid session
Return Codes for Control Variable	Access Functions:
MPI_T_CVAR_READ, WRITE	
MPI_T_ERR_CVAR_SETNOTNOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SETNEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALIDHANDLE	The handle is invalid
Return Codes for Performance Var	iable Access and Control:
MPI_T_PVAR_START, STOP, RI	EAD, WRITE, RESET, READRESET
MPI_T_ERR_INVALIDHANDLE	The handle is invalid
MPI_T_ERR_INVALIDSESSION	Session argument is not a valid session
MPI_T_ERR_PVAR_NOSTARTSTOP	Variable can not be started or stopped
	(for MPI_T_PVAR_START and
	MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NOWRITE	Variable can not be written or reset
	(for MPI_T_PVAR_WRITE and
	MPI_T_PVAR_RESET)
MPI_T_NOATOMIC	Variable cannot be read and written atomically
	(for MPI_T_PVAR_READRESET)
Return Codes for Category Function	
MPI_T_ERR_INVALIDINDEX	The category index is invalid

1

Chapter 15

Deprecated Functions

15.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HVECTOR in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)				
IN	IN count number of blocks (non-negative integer)			
IN	blocklength	number of elements in each block (non-negative inte-	24	
	bioenengen	ger)	25 26	
IN	stride	number of bytes between start of each block (integer)	20	
			28	
IN	oldtype	old datatype (handle)	29	
OUT	newtype	new datatype (handle)	30	
31				
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,				

For this routine, an interface within the mpi_f08 module was never defined.

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HINDEXED in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

Unofficial Draft for Comment Only

 $_{34}$ ticket 229.1.

	1 2	MPI_TYPI	E_HINDEXED(count, array_o type)	of_blocklengths, array_of_displacements, oldtype, new-
	3 4 5 6	IN	count	<pre>number of blocks - also number of entries in array_of_displacements and array_of_blocklengths (non- negative integer)</pre>
	7 8	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
	9	IN	array_of_displacements	byte displacement of each block (array of integer)
	10	IN	oldtype	old datatype (handle)
	11 12 13	OUT	newtype	new datatype (handle)
ticket229.1	14 15 16	int MPI_7		int *array_of_blocklengths, _displacements, MPI_Datatype oldtype, /pe)
ticket229.1	• 17 18	For this re	outine, an interface within th	ne mpi_f08 module was never defined.
	19 20 21 22	INTEC	OLDTYPE, NEWTYPE, 1	F_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, IERROR) KLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
	24 25 26 27 28	MPI_TYPI binding of	E_CREATE_STRUCT in MPI	cated and is superseded by -2.0. The language independent definition and the C ne same as of the new function, except of the function ling is different.
	29 30	MPI_TYPI	E_STRUCT(count, array_of_ newtype)	blocklengths, array_of_displacements, array_of_types,
	31 32 33 34	IN	count	number of blocks (integer) (non-negative integer) – also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
	35 36	IN	array_of_blocklength	number of elements in each block (array of non-negative integer)
	37	IN	array_of_displacements	byte displacement of each block (array of integer)
	38 39 40	IN	array_of_types	type of elements in each block (array of handles to datatype objects)
	40 41 42	OUT	newtype	new datatype (handle)
ticket229.1	43 44 45	int MPI_7	MPI_Aint *array_of_	nt *array_of_blocklengths, _displacements, y_of_types, MPI_Datatype *newtype)
	47 48	For this ro	outine, an interface within th	ne mpi_f08 module was never defined.

MPI_TYPE_	1 2 3					
<pre>INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR</pre>						
		and is superseded by $MPI_GET_ADDRESS$ in MPI_I	5 6			
		n and the C binding of the deprecated function is of the function name. Only the Fortran language	7 8			
	binding is different.					
			10			
MPI_ADD	RESS(location, address)		11 12			
IN	location	location in caller memory (choice)	13			
OUT	address	address of location (integer)	14			
			15			
int MPI_A	Address(void* location, M	PI_Aint *address)	$^{16}_{17}$ ticket229.1.			
For this re	outine, an interface within the	<pre>mpi_f08 module was never defined.</pre>	18			
MPI_ADDRE	ESS(LOCATION, ADDRESS, IER	RROR)	19			
<type< td=""><td><pre>> LOCATION(*)</pre></td><td></td><td>20</td></type<>	<pre>> LOCATION(*)</pre>		20			
INTEC	ER ADDRESS, IERROR		21 22			
The f	ollowing functions are depred	cated and are superseded by	23			
MPI_TYPI	24					
	25					
MPI_TYPI	26					
- IN	datatype	datatype (handle)	27 28			
OUT	extent	datatype (manus) datatype extent (integer)	29			
001	extent	datatype extent (integer)	30			
int MPT 1	vpe_extent(MPI_Datatype (datatype, MPI_Aint *extent)	31			
	32 ticket 229.1.					
For this ro	33					
	EXTENT(DATATYPE, EXTENT,		34 35			
INTEC	ER DATATYPE, EXTENT, IER	ROR	36			
Retur	ns the extent of a datatype, w	where extent is as defined on page 113 .	37			
		d for finding the lower bound and the upper bound	38			
of a dataty	vpe.		39			
			40 41			
MPI_TYPI	E_LB(datatype, displacement)		41 42			
IN	datatype	datatype (handle)	43			
OUT	displacement	displacement of lower bound from origin, in bytes (in-	44			
		teger)	45			
			46			
int MPI_7	47 48					

CHAPTER 15. DEPRECATED FUNCTIONS

ticket229.1.

	1					
	2	For this ro	utine, an interface wit	thin the mpi_f08 module was never defined.		
	3 4	MPI_TYPE_	LB(DATATYPE, DISP	LACEMENT, IERROR)		
	4 5	INTEG	ER DATATYPE, DISPL	ACEMENT, IERROR		
	6					
	7 8	MPI_TYPE	_UB(datatype, displa	cement)		
	9	IN	datatype	datatype (handle)		
	10	OUT	displacement	displacement of upper bound from origin, in bytes (in-		
	11			teger)		
	12 13					
ticket229.1		<pre>int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)</pre>				
	15 16	For this routine, an interface within the $\mathtt{mpi_f08}$ module was never defined.				
	17	MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERROR)				
	18	INTEG	ER DATATYPE, DISPL	ACEMENT, IERROR		
	19		0	deprecated and is superseded by		
	20 21			in MPI-2.0. The language independent definition of the		
	22	*		as that of the new function, except for the function name C/Fortran language interoperability see Section 16.3.7 on		
	23	and a different behavior in the C/Fortran language interoperability, see Section 16.3.7 on page 704. The language bindings are modified.				
	24					
	25 26	MPI_KEYV	/AL_CREATE(copy_fn	, delete_fn, keyval, extra_state)		
	27	IN	copy_fn	Copy callback function for keyval		
	28 29	IN	delete_fn	Delete callback function for keyval		
	30	OUT	keyval	key value for future access (integer)		
	31	IN	extra_state	Extra state for callback functions		
	32 33					
	34	int MPI_K	•	<pre>copy_function *copy_fn, MPI_Delete_function nt *keyval, void* extra_state)</pre>		
ticket 229.1	. 35	D .11		·		
	36 37	For this rol	utine, an interface wit	thin the mpi_f08 module was never defined.		
	38			DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)		
	39		NAL COPY_FN, DELET ER KEYVAL, EXTRA_S			
	40		-			
	41 42			voked when a communicator is duplicated by ld be of type MPI_Copy_function, which is defined as follows:		
	43	_	15-			
	44	typedef i	nt MPI Copy functi	on(MPI_Comm oldcomm, int keyval,		
	45 46	Speace 1		void *extra_state, void *attribute_val_in,		
	40			<pre>void *attribute_val_out, int *flag)</pre>		
	48					

ticket229.1.	A Fortran declaration for such a function is as follows:	1 2			
	For this routine, an interface within the mpi_f08 module was never defined.				
	SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	3 4			
	ATTRIBUTE_VAL_OUT, FLAG, IERR)	5			
	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	6			
	ATTRIBUTE_VAL_OUT, IERR	7			
	LOGICAL FLAG	8			
	copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or	9			
	FORTRAN; MPI_NULL_COPY_FN is a function that does nothing other than returning	10			
	$flag = 0$ and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets $flag =$	11			
	1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note	12			
	that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.	13			
	Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn	14			
	function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call	15			
	is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function,	16			
	which is defined as follows:	17			
		18			
	typedef int MPI_Delete_function(MPI_Comm comm, int keyval,	19			
	void *attribute_val, void *extra_state);	20			
	······································	21			
	A Fortran declaration for such a function is as follows:				
	For this routine, an interface within the mpi_f08 module was never defined.	23			
		24			
	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)	25			
	INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR				
	$delete_fn$ may be specified as $MPI_NULL_DELETE_FN$ from either C or FORTRAN				
	MPI_NULL_DELETE_FN is a function that does nothing, other than returning	28			
	MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.	29			
	The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL	30			
	in MPI-2.0. The language independent definition of the deprecated function is the same as	31			
	of the new function, except of the function name. The language bindings are modified.	32			
		33			
		34			
	MPI_KEYVAL_FREE(keyval)	35			
	INOUT keyval Frees the integer key value (integer)	36			
		37			
	int MPI_Keyval_free(int *keyval)	38			
	Int Hri_Keyvai_Ilee(Int *Keyval)	³⁹ ticket229.1.			
	For this routine, an interface within the mpi_f08 module was never defined.	40			
	MPI_KEYVAL_FREE(KEYVAL, IERROR)	41			
	INTEGER KEYVAL, IERROR	42			
	Initial and a filler of the second seco	43			
	The following function is deprecated and is superseded by $MPI_COMM_SET_ATTR$ in	44			
	MPI-2.0. The language independent definition of the deprecated function is the same as of	45			
	the new function, except of the function name. The language bindings are modified.	46			
		47			
		48			

	1	MPI_ATTR	_PUT(comm, keyval, attribute	_val)
	2 3	INOUT	comm	communicator to which attribute will be attached (han- dle)
	4 5 6	IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)
	7 8	IN	attribute_val	attribute value
ticket229.1	9 10	int MPI_A	ttr_put(MPI_Comm comm, in	t keyval, void* attribute_val)
UICKet229.1	11	For this rou	tine, an interface within the	mpi_f08 module was never defined.
	12 13 14		PUT(COMM, KEYVAL, ATTRIBU ER COMM, KEYVAL, ATTRIBUT	
	15 16 17 18	MPI-2.0. T	he language independent defin	and is superseded by MPI_COMM_GET_ATTR in nition of the deprecated function is the same as of name. The language bindings are modified.
	19 20	MPI_ATTR	_GET(comm, keyval, attribute	_val, flag)
	20 21	IN	comm	communicator to which attribute is attached (handle)
	22	IN	keyval	key value (integer)
	23 24	OUT	attribute_val	attribute value, unless $flag = false$
	25 26 27	OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key
ticket229.1	28	int MPI_A	ttr_get(MPI_Comm comm, in	t keyval, void *attribute_val, int *flag)
010100220.1	30	For this rou	itine, an interface within the	mpi_f08 module was never defined.
	31 32 33 34	INTEG	GET(COMM, KEYVAL, ATTRIBU ER COMM, KEYVAL, ATTRIBUT AL FLAG	
	35 36 37 38	in MPI-2.0.	The language independent d	and is superseded by MPI_COMM_DELETE_ATTR efinition of the deprecated function is the same as on name. The language bindings are modified.
	39 40	MPI_ATTR	_DELETE(comm, keyval)	
	40 41	INOUT	comm	communicator to which attribute is attached (handle)
	42	IN	keyval	The key value of the deleted attribute (integer)
	43 44	int MDT A	ttr_delete(MPI_Comm comm,	int keywal)
ticket229.1				•
	46 47			mpi_f08 module was never defined.
	48	MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)		

	INTEG	ER COMM, KEYVAL, IERROR		1			
	2						
	3						
	MPI_COMM_CREATE_ERRHANDLER in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name.						
	The language bindings are modified.						
				6 7			
	MPI_ERRH	IANDLER_CREATE([function]handler_fn, errhandler)	$^{8}_{9}$ ticket252-W.			
	IN	[ticket252-W.][function]hand	er_fn user defined error handling procedure	9 10			
	OUT	errhandler	MPI error handler (handle)	11			
	001		wittenormandier (nandie)	12			
	int MPI_E	rrhandler_create(MPI_Hand MPI_Errhandler *errh	dler_function *[function]handler_fn,	$^{13}_{14}$ ticket252-W.			
		MPI_EIIMANGIEI *eIIM		$_{15}$ ticket 229.1.			
	For this ro	utine, an interface within the	<pre>mpi_f08 module was never defined.</pre>	16			
	MPI ERRHA	NDLER CREATE([FUNCTION]H	ANDLER_FN, ERRHANDLER, IERROR)	17 ticket252-W.			
		NAL [FUNCTION] HANDLER_FN		18 ticket252-W.			
	INTEG	ER ERRHANDLER, IERROR		19			
	Dorist	or the user routing [function	handler_fn for use as an MPI exception handler.	$^{20}_{21}$ ticket252-W.			
	0		· · ·	21 UICKet252- W.			
Returns in errhandler a handle to the registered exception handler. In the C language, the user routine should be a C function of type MPI_Handler_function,							
	which is de			23 24			
				25			
	typedef v	oid (MPI_Handler_function	n)(MPI_Comm *, int *,);	26			
	TTI C		• • • • • • • • • • • • • • • • •	27			
		rst argument is the commun	nicator in use, the second is the error code to be	28			
	returned.	29					
	III the	Fortran language, the user re	Suthe should be of the form.	30			
	SUBROUTIN	E HANDLER_FUNCTION(COMM,	ERROR_CODE)	31			
		R COMM, ERROR_CODE		32			
				33 34			
		ollowing function is depreca	· ·	35			
			PI-2.0. The language independent definition of the	36			
	-		ne new function, except of the function name. The	37			
	language c	indings are modified.		38			
				39			
	MPI_ERRH	IANDLER_SET(comm, errhar	ndler)	40			
	INOUT	comm	communicator to set the error handler for (handle)	41			
		errhandler		42			
	IN	ermandier	new MPI error handler for communicator (handle)	43			
	int MDT T	mehandlan aat (MDT Carry -	omm MDI Errhondlor orrhondlor	44			
	IIIC MPI_E	TINGUATEL SET (MP1_COMM C	omm, MPI_Errhandler errhandler)	$^{45}_{46}$ ticket 229.1.			
	For this ro	utine, an interface within the	<pre>mpi_f08 module was never defined.</pre>	40			
	MPI_ERRHA	NDLER_SET(COMM, ERRHANDL	ER, IERROR)	48			
		_ 、 _ ,					

Unofficial Draft for Comment Only

	1	INTEGER COMM, ERRHANDLER, IERF	OR			
	2 3 4	process. Note that an error handler is al				
	5 6 7		ed and is superseded by PI-2.0. The language independent definition of the e new function, except of the function name. The			
	8 9	language bindings are modified.				
	10 11	MPI_ERRHANDLER_GET(comm, errhan	dler)			
	11	IN comm	communicator to get the error handler from (handle)			
	13 14 15	OUT errhandler	MPI error handler currently associated with communicator (handle)			
ticket229.1	16	<pre>int MPI_Errhandler_get(MPI_Comm co</pre>	mm, MPI_Errhandler *errhandler)			
	18	For this routine, an interface within the mpi_f08 module was never defined.				
	19 20 21	MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR				
	21 22 23	Returns in errhandler (a handle to) the error handler that is currently associated with communicator comm.				
	24					
	25 26	15.2 Deprecated since MPI-2.2				
	27 28	The entire set of C++ language bindings have been deprecated.				
	29	Rationale. The C++ bindings add minimal functionality over the C bindings while				
	30	incurring a significant amount of maintenance to the MPI specification. Since the $C + 1$ bindings are effectively a one to one mapping of the C bindings, it should be				
	31	C++ bindings are effectively a one-to-one mapping of the C bindings, it should be relatively easy to convert existing $C++$ MPI applications to use the MPI C bindings.				
	32 33	relatively easy to convert existing $C++$ MPI applications to use the MPI C bindings. Additionally, there are third party packages available that provide $C++$ class library				
	34	functionality (i.e., $C++$ -specific functionality layered on top of the MPI C bindings)				
	35	that are likely more expressive and/or natural to $C++$ programmers and are not				
	36	suitable for standardization in this	specification. (End of rationale.)			
	37	The following function typedefs ha	we been deprecated and are superseded by new			
	38 39		the function signatures are exactly the same; the			
	40	names were updated to match conventio				
	41					
	42	Deprecated Name	New Name			
	43	MPI_Comm_errhandler_fn MPI::Comm::Errhandler_fn	MPI_Comm_errhandler_function MPI::Comm::Errhandler_function			
	44	mrioummiErrmanuter_In	milloommErrmanuter_runction			
		MPI File errhandler fn	MPI File errhandler function			
	45	MPI_File_errhandler_fn MPI::File::Errhandler_fn	MPI_File_errhandler_function MPI::File::Errhandler_function			
		MPI_File_errhandler_fn MPI::File::Errhandler_fn MPI_Win_errhandler_fn				

15.3 Deprecated since MPI-3.0

[]

Chapter 16

Language Bindings

16.1 C++

16.1.1 Overview

The C++ language bindings have been deprecated. A compliant MPI implementation providing C++ language bindings must provide the entire set defined in this document.

There are some issues specific to C++ that must be considered in the design of an interface that go beyond the simple description of language bindings. In particular, in C++, we must be concerned with the design of objects and their interfaces, rather than just the design of a language-specific functional interface to MPI. Fortunately, the design of MPI was based on the notion of objects, so a natural set of classes is already part of MPI.

MPI-2 includes C++ bindings as part of its function specifications. In some cases, MPI-2 provides new names for the C bindings of MPI-1 functions. In this case, the C++binding matches the new C name — there is no binding for the deprecated name.

16.1.2 Design

The C++ language interface for MPI is designed according to the following criteria:

- 1. The C++ language interface consists of a small set of classes with a lightweight functional interface to MPI. The classes are based upon the fundamental MPI object types (e.g., communicator, group, etc.).
- 2. The MPI C++ language bindings provide a semantically correct interface to MPI.
- 3. To the greatest extent possible, the C++ bindings for MPI functions are member functions of MPI classes.

Rationale. Providing a lightweight set of MPI objects that correspond to the basic MPI types is the best fit to MPI's implicit object-based design; methods can be supplied for these objects to realize MPI functionality. The existing C bindings can be used in C++ programs, but much of the expressive power of the C++ language is forfeited. On the other hand, while a comprehensive class library would make user programming more elegant, such a library it is not suitable as a language binding for MPI since a binding must provide a direct and unambiguous mapping to the specified functionality of MPI. (*End of rationale.*)

18 ticket279.

 24

 31

16.1.3 C++ Classes for MPI

All MPI classes, constants, and functions are declared within the scope of an MPI namespace. Thus, instead of the MPI_ prefix that is used in C and Fortran, MPI functions essentially have an MPI:: prefix.

The members of the MPI namespace are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the MPI namespace and its member classes is as follows:

```
namespace MPI {
10
        class Comm
                                                     \{...\};
11
        class Intracomm : public Comm
                                                     \{...\}:
12
        class Graphcomm : public Intracomm
                                                     \{...\};
13
        class Distgraphcomm : public Intracomm {...};
14
        class Cartcomm : public Intracomm
                                                     \{...\};
15
        class Intercomm : public Comm
                                                     \{...\};
16
        class Datatype
                                                     \{...\};
17
                                                     \{...\};
        class Errhandler
18
                                                     \{...\};
        class Exception
19
        class File
                                                     \{...\};
20
                                                     \{...\};
        class Group
21
        class Info
                                                     \{...\};
22
        class Op
                                                     \{...\};
23
        class Request
                                                     \{...\};
24
                                                     \{...\};
        class Prequest
                         : public Request
25
        class Grequest : public Request
                                                     \{...\};
26
        class Status
                                                     \{...\};
27
                                                     \{\ldots\};
        class Win
28
     };
29
```

Note that there are a small number of derived classes, and that virtual inheritance is *not* used.

16.1.4 Class Member Functions for MPI

Besides the member functions which constitute the C++ language bindings for MPI, the C++ language interface has additional functions (as required by the C++ language). In particular, the C++ language interface must provide a constructor and destructor, an assignment operator, and comparison operators.

The complete set of C++ language bindings for MPI is presented in Annex A.5. The bindings take advantage of some important C++ features, such as references and const. Declarations (which apply to all MPI member classes) for construction, destruction, copying, assignment, comparison, and mixed-language operability are also provided.

Except where indicated, all non-static member functions (except for constructors and
 the assignment operator) of MPI member classes are virtual functions.

Rationale. Providing virtual member functions is an important part of design for
 inheritance. Virtual functions can be bound at run-time, which allows users of libraries
 to re-define the behavior of objects already contained in a library. There is a small

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8 9

30

 31

32 33

34

performance penalty that must be paid (the virtual function must be looked up before it can be called). However, users concerned about this performance penalty can force compile-time function binding. (*End of rationale.*)

Example 16.1 Example showing a derived MPI class.

Advice to implementors. Implementors must be careful to avoid unintended side effects from class libraries that use inheritance, especially in layered implementations. For example, if MPI_BCAST is implemented by repeated calls to MPI_SEND or MPI_RECV, the behavior of MPI_BCAST cannot be changed by derived communicator classes that might redefine MPI_SEND or MPI_RECV. The implementation of MPI_BCAST must explicitly use the MPI_SEND (or MPI_RECV) of the base MPI:::Comm class. (End of advice to implementors.)

16.1.5 Semantics

The semantics of the member functions constituting the C++ language binding for MPI are specified by the MPI function description itself. Here, we specify the semantics for those portions of the C++ language interface that are not part of the language binding. In this subsection, functions are prototyped using the type MPI:: $\langle CLASS \rangle$ rather than listing each function for every MPI class; the word $\langle CLASS \rangle$ can be replaced with any valid MPI class name (e.g., Group), except as noted.

Construction / Destruction The default constructor and destructor are prototyped as follows:

```
{ MPI::<<CLASS>() (binding deprecated, see Section 15.2) }
```

{ ~MPI::<CLASS>() (binding deprecated, see Section 15.2) }

In terms of construction and destruction, opaque MPI user level objects behave like handles. Default constructors for all MPI objects except MPI::Status create corresponding MPI::*_NULL handles. That is, when an MPI object is instantiated, comparing it with its corresponding MPI::*_NULL object will return true. The default constructors do not create new MPI opaque objects. Some classes have a member function Create() for this purpose.

Example 16.2 In the following code fragment, the test will return **true** and the message will be sent to **cout**.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 31

 $45 \\ 46$

```
1
     void foo()
\mathbf{2}
     {
3
       MPI::Intracomm bar;
4
5
        if (bar == MPI::COMM NULL)
6
          cout << "bar is MPI::COMM_NULL" << endl;</pre>
7
     }
8
9
          The destructor for each MPI user level object does not invoke the corresponding
     MPI_*_FREE function (if it exists).
10
11
                        MPI_*_FREE functions are not automatically invoked for the following
           Rationale.
12
           reasons:
13
14
             1. Automatic destruction contradicts the shallow-copy semantics of the MPI classes.
15
             2. The model put forth in MPI makes memory allocation and deallocation the re-
16
                sponsibility of the user, not the implementation.
17
18
             3. Calling MPI_*_FREE upon destruction could have unintended side effects, in-
19
                cluding triggering collective operations (this also affects the copy, assignment,
20
                and construction semantics). In the following example, we would want neither
21
                foo_comm nor bar_comm to automatically invoke MPI_*_FREE upon exit from
22
                the function.
23
                void example_function()
24
                ſ
25
                  MPI::Intracomm foo_comm(MPI::COMM_WORLD), bar_comm;
26
                  bar_comm = MPI::COMM_WORLD.Dup();
27
                  // rest of function
28
                }
29
30
           (End of rationale.)
^{31}
32
     Copy / Assignment The copy constructor and assignment operator are prototyped as fol-
33
     lows:
34
     { MPI:::<CLASS>(const MPI:::<CLASS>& data) (binding deprecated, see Section 15.2) }
35
36
     { MPI:::<CLASS>& MPI:::<CLASS>::operator=(const MPI:::<CLASS>& data) (binding
37
                     deprecated, see Section 15.2 }
38
          In terms of copying and assignment, opaque MPI user level objects behave like handles.
39
     Copy constructors perform handle-based (shallow) copies. MPI::Status objects are excep-
40
     tions to this rule. These objects perform deep copies for assignment and copy construction.
41
42
           Advice to implementors.
                                      Each MPI user level object is likely to contain, by value
43
           or by reference, implementation-dependent state information. The assignment and
44
           copying of MPI object handles may simply copy this value (or reference). (End of
45
           advice to implementors.)
46
47
48
```

Example 16.3 Example using assignment operator. In this example, MPI::Intracomm::Dup() is not called for foo_comm. The object foo_comm is simply an alias for MPI::COMM_WORLD. But bar_comm is created with a call to MPI::Intracomm::Dup() and is therefore a different communicator than foo_comm (and thus different from MPI::COMM_WORLD). baz_comm becomes an alias for bar_comm. If one of bar_comm or baz_comm is freed with MPI_COMM_FREE it will be set to MPI::COMM_NULL. The state of the other handle will be undefined — it will be invalid, but not necessarily set to MPI::COMM_NULL.

MPI::Intracomm foo_comm, bar_comm, baz_comm;
<pre>foo_comm = MPI::COMM_WORLD; bar_comm = MPI::COMM_WORLD.Dup(); baz_comm = bar_comm;</pre>
Comparison The comparison operators are prototyped as follows:
<pre>{bool MPI::<class>::operator==(const MPI::<class>& data) const(binding</class></class></pre>
<pre>{bool MPI::<class>::operator!=(const MPI::<class>& data) const(binding</class></class></pre>

The member function operator==() returns true only when the handles reference the same internal MPI object, false otherwise. operator!=() returns the boolean complement of operator==(). However, since the Status class is not a handle to an underlying MPI object, it does not make sense to compare Status instances. Therefore, the operator==() and operator!=() functions are not defined on the Status class.

Constants Constants are singleton objects and are declared const. Note that not all globally defined MPI objects are constant. For example, MPI::COMM_WORLD and MPI::COMM_SELF are not const.

16.1.6 C++ Datatypes

Table 16.1 lists all of the C++ predefined MPI datatypes and their corresponding C and C++ datatypes, Table 16.2 lists all of the Fortran predefined MPI datatypes and their corresponding Fortran 77 datatypes. Table 16.3 lists the C++ names for all other MPI datatypes.

MPI::BYTE and MPI::PACKED conform to the same restrictions as MPI_BYTE and MPI_PACKED, listed in Sections 3.2.2 on page 29 and Sections 4.2 on page 140, respectively.

The following table defines groups of MPI predefined datatypes:

C integer:	MPI::INT, MPI::LONG, MPI::SHORT,	42
-	MPI::UNSIGNED_SHORT, MPI::UNSIGNED,	43
	MPI::UNSIGNED_LONG,	44
	MPI::_LONG_LONG, MPI::UNSIGNED_LONG_L	ONG,
	MPI::SIGNED_CHAR, MPI::UNSIGNED_CHAR	46
Fortran integer:	MPI::INTEGER	47
	and handles returned from	48

Unofficial Draft for Comment Only

 24

Unofficial Draft for Comment Only

MPI datatype	Description
MPI::FLOAT_INT	C/C++ reduction type
MPI::DOUBLE_INT	C/C++ reduction type
MPI::LONG_INT	C/C++ reduction type
MPI::TWOINT	C/C++ reduction type
MPI::SHORT_INT	C/C++ reduction type
MPI::LONG_DOUBLE_INT	C/C++ reduction type
MPI::TWOREAL	Fortran reduction type
MPI::TWODOUBLE_PRECISION	Fortran reduction type
MPI::TWOINTEGER	Fortran reduction type
MPI::F_DOUBLE_COMPLEX	Optional Fortran type
MPI::INTEGER1	Explicit size type
MPI::INTEGER2	Explicit size type
MPI::INTEGER4	Explicit size type
MPI::INTEGER8	Explicit size type
MPI::INTEGER16	Explicit size type
MPI::REAL2	Explicit size type
MPI::REAL4	Explicit size type
MPI::REAL8	Explicit size type
MPI::REAL16	Explicit size type
MPI::F_COMPLEX4	Explicit size type
MPI::F_COMPLEX8	Explicit size type
MPI::F_COMPLEX16	Explicit size type
MPI::F_COMPLEX32	Explicit size type

Table 16.3: C++ names for other MPI data types. Implementations may also define other optional types (e.g., MPI::INTEGER8).

1		MPI::Datatype::Create_f90_integer,
2		and if available: MPI::INTEGER1,
3		MPI::INTEGER2, MPI::INTEGER4,
4		MPI::INTEGER8, MPI::INTEGER16
-	Floating point:	MPI::FLOAT, MPI::DOUBLE, MPI::REAL,
5	Floating point:	
6		MPI::DOUBLE_PRECISION,
7		MPI::LONG_DOUBLE
8		and handles returned from
9		MPI::Datatype::Create_f90_real,
10		and if available: MPI::REAL2,
11		MPI::REAL4, MPI::REAL8, MPI::REAL16
12	Logical:	MPI::LOGICAL, MPI::BOOL
	Complex:	MPI::F_COMPLEX, MPI::COMPLEX,
13	complex.	MPI::F_DOUBLE_COMPLEX,
14		MPI::DOUBLE_COMPLEX,
15		,
16		MPI::LONG_DOUBLE_COMPLEX
17		and handles returned from
18		MPI::Datatype::Create_f90_complex,
		and if available: MPI::F_DOUBLE_COMPLEX,
19		MPI::F_COMPLEX4, MPI::F_COMPLEX8,
20		MPI::F_COMPLEX16, MPI::F_COMPLEX32
21	Byte:	MPI::BYTE
22	-	
23		eration are specified below in terms of the groups
24	defined above.	
25		
26		
	Ор	Allowed Types
27		U X
28	MPI::MAX, MPI::MIN	C integer, Fortran integer, Floating point
29	MPI::SUM, MPI::PROD	C integer, Fortran integer, Floating point, Complex
30	MPI::LAND, MPI::LOR, MPI::LXOR	C integer, Logical
31		
32	MPI::BAND, MPI::BOR, MPI::BXOR	C integer, Fortran integer, Byte
33	MPI::MINLOC and MPI::MAXLOC perfor	rm just as their C and Fortran counterparts; see
34	Section 5.9.4 on page 191.	J
	Section 5.5.1 on page 101.	
35		
36	16.1.7 Communicators	
37	The MDI Comm class biorershy makes even	icit the different kinds of communicators implic-
38	· · · ·	· · · · · · · · · · · · · · · · · · ·
39		strongly typed. Since the original design of MPI
40		es of communicators, the following clarifications
41	are provided for the C++ design.	
42		
	Types of communicators There are six dif	ferent types of communicators: MPI::Comm,
43	MPI::Intercomm, MPI::Intracomm, MPI:	, , , , , , , , , , , , , , , , , , ,
44	, , , , , , , , , , , , , , , , , , , ,	· • •
45		abstract base communicator class, encapsulating
46	the functionality common to all MPI com	nmunicators. MPI:::Intercomm and
47		omm. MPI:::Cartcomm, MPI::Graphcomm, and
47 48		
	MPI:::Intracomm are derived from MPI::C	

1 Advice to users. Initializing a derived class with an instance of a base class is 2 ticket182. not [legal]valid in C++. For instance, it is not [legal]valid to initialize a Cartcomm ticket182. 3 from an Intracomm. Moreover, because MPI::Comm is an abstract base class, it is 4 non-instantiable, so that it is not possible to have an object of class MPI::Comm. However, it is possible to have a reference or a pointer to an MPI::Comm. 56 7 **Example 16.4** The following code is erroneous. 8 9 Intracomm intra = MPI::COMM_WORLD.Dup(); 10 Cartcomm cart(intra); // This is erroneous 11 (End of advice to users.) 1213 14MPI::COMM_NULL The specific type of MPI::COMM_NULL is implementation dependent. 15MPI::COMM_NULL must be able to be used in comparisons and initializations with all types 16of communicators. MPI::COMM_NULL must also be able to be passed to a function that 17expects a communicator argument in the parameter list (provided that MPI::COMM_NULL 18 is an allowed value for the communicator argument). 19 There are several possibilities for implementation of MPI::COMM_NULL. Rationale. 20Specifying its required behavior, rather than its realization, provides maximum flexi-21bility to implementors. (End of rationale.) 22 23 24 **Example 16.5** The following example demonstrates the behavior of assignment and com-25parison using MPI::COMM_NULL. 26MPI:::Intercomm comm; 27comm = MPI::COMM_NULL; // assign with COMM_NULL 2829 if (comm == MPI::COMM_NULL) // true 30 cout << "comm is NULL" << endl;</pre> if (MPI::COMM_NULL == comm) 31// note -- a different function! cout << "comm is still NULL" << endl;</pre> 32 33 Dup() is not defined as a member function of MPI::Comm, but it is defined for the 34 derived classes of MPI::Comm. Dup() is not virtual and it returns its OUT parameter by 35 value. 36 37 MPI::Comm::Clone() The C++ language interface for MPI includes a new function 38 Clone(). MPI::Comm::Clone() is a pure virtual function. For the derived communicator 39 classes, Clone() behaves like Dup() except that it returns a new object by reference. The 40 Clone() functions are prototyped as follows: 41 Comm& Comm::Clone() const = 0 4243 Intracomm& Intracomm::Clone() const 44Intercomm& Intercomm::Clone() const 4546Cartcomm& Cartcomm::Clone() const 4748

Unofficial Draft for Comment Only

Graphcomm& Graphcomm::Clone() const

1	Distgraphcomm& Distgraphcomm::Clone() const
2	
3	Rationale. Clone() provides the "virtual dup" functionality that is expected by C++
4	programmers and library writers. Since Clone() returns a new object by reference,
5	users are responsible for eventually deleting the object. A new name is introduced
6 7	rather than changing the functionality of Dup(). (End of rationale.)
8	
9	Advice to implementors. Within their class declarations, prototypes for Clone() and
10	Dup() would look like the following:
11	namespace MPI {
12	class Comm {
13	<pre>virtual Comm& Clone() const = 0;</pre>
14	};
15	class Intracomm : public Comm {
16	<pre>Intracomm Dup() const { };</pre>
17	<pre>virtual Intracomm& Clone() const { };</pre>
18 19	}; class Intercomm : public Comm {
20	Intercomm Dup() const { };
21	virtual Intercomm& Clone() const { };
22	};
23	// Cartcomm, Graphcomm,
24	<pre>// and Distgraphcomm are similarly defined</pre>
25	};
26	(End of advice to implementors.)
27	
28 29	16.1.8 Exceptions
30	
31	The C++ language interface for MPI includes the predefined error handler
32	MPI::ERRORS_THROW_EXCEPTIONS for use with the Set_errhandler() member functions. MPI::ERRORS_THROW_EXCEPTIONS can only be set or retrieved by C++ functions. If a
33	non-C++ program causes an error that invokes the MPI:::ERRORS_THROW_EXCEPTIONS error
34	handler, the exception will pass up the calling stack until $C++$ code can catch it. If there
35	is no C++ code to catch it, the behavior is undefined. In a multi-threaded environment
36	or if a nonblocking MPI call throws an exception while making progress in the background,
37	the behavior is implementation dependent.
38	The error handler MPI:::ERRORS_THROW_EXCEPTIONS causes an MPI::Exception to be
39 40	thrown for any MPI result code other than MPI::SUCCESS. The public interface to
41	MPI::Exception class is defined as follows:
42	namespace MPI {
43	class Exception {
44	public:
45	-
46	<pre>Exception(int error_code);</pre>
47	
48	<pre>int Get_error_code() const;</pre>

```
int Get_error_class() const;
  const char *Get_error_string() const;
};
};
```

Advice to implementors.

The exception will be thrown within the body of MPI:::ERRORS_THROW_EXCEPTIONS. It is expected that control will be returned to the user when the exception is thrown. Some MPI functions specify certain return information in their parameters in the case of an error and MPI_ERRORS_RETURN is specified. The same type of return information must be provided when exceptions are thrown.

For example, MPI_WAITALL puts an error code for each request in the corresponding entry in the status array and returns MPI_ERR_IN_STATUS. When using MPI::ERRORS_THROW_EXCEPTIONS, it is expected that the error codes in the status array will be set appropriately before the exception is thrown.

(End of advice to implementors.)

16.1.9 Mixed-Language Operability

The C++ language interface provides functions listed below for mixed-language operability. These functions provide for a seamless transition between C and C++. For the case where the C++ class corresponding to <CLASS> has derived classes, functions are also provided for converting between the derived classes and the C MPI_<CLASS>.

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
```

MPI::<CLASS>(const MPI_<CLASS>& data)

MPI::<CLASS>::operator MPI_<CLASS>() const

These functions are discussed in Section 16.3.4.

16.1.10 Profiling

This section specifies the requirements of a C++ profiling interface to MPI.

Advice to implementors. Since the main goal of profiling is to intercept function calls from user code, it is the implementor's decision how to layer the underlying implementation to allow function calls to be intercepted and profiled. If an implementation of the MPI C++ bindings is layered on top of MPI bindings in another language (such as C), or if the C++ bindings are layered on top of a profiling interface in another language, no extra profiling interface is necessary because the underlying MPI implementation already meets the MPI profiling interface requirements.

Native C++MPI implementations that do not have access to other profiling interfaces must implement an interface that meets the requirements outlined in this section.

High-quality implementations can implement the interface outlined in this section in order to promote portable C++ profiling libraries. Implementors may wish to provide an option whether to build the C++ profiling interface or not; C++ implementations that are already layered on top of bindings in another language or another profiling 48

Unofficial Draft for Comment Only

1

2

3

 $\frac{4}{5}$

6 7

8

9

10

11

12

13

14

15

16

17 18

19

20

21

22

23

24 25

26

27 28

29

30 31

32 33

34 35

36

37

38

39

40

41

42

43

1 2	interface will have to insert a third layer to implement the $C++$ profiling interface. (End of advice to implementors.)
3 4 5	To meet the requirements of the C++ MPI profiling interface, an implementation of the MPI functions <i>must</i> :
6 7 8 9	 Provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix "MPI::") should also be accessible with the prefix "PMPI::."
10 11	2. Ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.
12 13 14 15 16	3. Document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that profiler developer knows whether they must implement the profile interface for each binding, or can economize by imple- menting it only for the lowest level routines.
17 18 19 20	4. Where the implementation of different language bindings is done through a layered approach (e.g., the C++ binding is a set of "wrapper" functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.
21 22 23 24 25 26	This is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the author of the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.
27	5. Provide a no-op routine MPI::Pcontrol in the MPI library.
28 29 30 31 32 33	Advice to implementors. There are (at least) two apparent options for implementing the C++ profiling interface: inheritance or caching. An inheritance-based approach may not be attractive because it may require a virtual inheritance implementation of the communicator classes. Thus, it is most likely that implementors will cache PMPI objects on their corresponding MPI objects. The caching scheme is outlined below.
34 35	The "real" entry points to each routine can be provided within a namespace PMPI. The non-profiling version can then be provided within a namespace MPI.
36 37 38	Caching instances of PMPI objects in the MPI handles provides the "has a" relationship that is necessary to implement the profiling scheme.
39 40 41	Each instance of an MPI object simply "wraps up" an instance of a PMPI object. MPI objects can then perform profiling actions before invoking the corresponding function in their internal PMPI object.
42 43 44 45 46	The key to making the profiling work by simply re-linking programs is by having a header file that <i>declares</i> all the MPI functions. The functions must be <i>defined</i> elsewhere, and compiled into a library. MPI constants should be declared extern in the MPI namespace. For example, the following is an excerpt from a sample mpi.h file:
47 48	Example 16.6 Sample mpi.h file.

```
1
namespace PMPI {
                                                                                            \mathbf{2}
  class Comm {
                                                                                            3
  public:
    int Get_size() const;
                                                                                            4
                                                                                            5
  }:
                                                                                            6
  // etc.
                                                                                            7
};
                                                                                            8
                                                                                            9
namespace MPI {
                                                                                            10
public:
                                                                                            11
  class Comm {
  public:
                                                                                            12
                                                                                            13
     int Get_size() const;
                                                                                            14
                                                                                            15
  private:
                                                                                            16
    PMPI::Comm pmpi_comm;
                                                                                            17
  };
                                                                                            18
};
                                                                                            19
```

Note that all constructors, the assignment operator, and the destructor in the MPI class will need to initialize/destroy the internal PMPI object as appropriate.

The definitions of the functions must be in separate object files; the PMPI class member functions and the non-profiling versions of the MPI class member functions can be compiled into libmpi.a, while the profiling versions can be compiled into libmpi.a. Note that the PMPI class member functions and the MPI constants must be in different object files than the non-profiling MPI class member functions in the libmpi.a library to prevent multiple definitions of MPI class member function names when linking both libmpi.a and libpmpi.a. For example:

```
Example 16.7 pmpi.cc, to be compiled into libmpi.a.
```

```
int PMPI::Comm::Get_size() const
{
    // Implementation of MPI_COMM_SIZE
}
```

Example 16.8 constants.cc, to be compiled into libmpi.a.

```
const MPI::Intracomm MPI::COMM_WORLD;
```

Example 16.9 mpi_no_profile.cc, to be compiled into libmpi.a.

```
int MPI::Comm::Get_size() const
{
    return pmpi_comm.Get_size();
}
```

20

21 22

23

 24

25

26

27

28

29 30

31 32

33

34

35

36 37 38

39 40

 $41 \\ 42$

43 44

 $45 \\ 46$

47

1	Example 16.10 mpi_profile.cc , to be compiled into libpmpi.a.
2 3	<pre>int MPI::Comm::Get_size() const</pre>
4	{
5	// Do profiling stuff
6	<pre>int ret = pmpi_comm.Get_size();</pre>
7	// More profiling stuff
8	return ret;
9	}
10	
11 12	(End of advice to implementors.)
13	16.2 Fortron Support
14	16.2 Fortran Support
15 16	16.2.1 Overview
ticket230-B. 17	The Fortran [MPI-2]MPI language bindings have been designed to be compatible with the
ticket230-B. 18	Fortran 90 standard [(and later)] with additional features from Fortran 2003 and Fortran
ticket230-B. 19	2008 [39] + TR 29113 [41]. [These bindings are in most cases compatible with Fortran 77,
20	implicit-style interfaces.]
21	
22	Rationale. Fortran 90 contains numerous features designed to make it a more "mod-
ticket0. 23	ern" language than Fortran 77. It seems natural that [MPI]MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90.
24 ticket230-B. 25	[MPI does not (yet) use many of these features because of a number of technical dif-
26	ficulties.]In Fortran $2008 + TR 29113$, the major new language features used are the
20	ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type
28	and assumed-rank dummy arguments for choice buffer arguments. Further require-
29	ments for compiler support are listed in Section 16.2.7 on page 659. (End of rationale.)
30	
ticket 230-B. $_{31}$	MPI defines [two levels]three methods of Fortran support[, described in Sections 16.2.4
ticket 230-B. $_{32}$	and 16.2.3. In the rest of this section, "Fortran" and "Fortran 90" shall refer to "Fortran
ticket 230-B. $_{33}$	90" and its successors, unless qualified.]:
34	1 USE mpi 608. This method is described in Section 16.2.2 and requires compile
35	1. USE mpi_f08: This method is described in Section 16.2.2 and requires compile- time argument checking with unique MPI handle types and provides techniques to
$_{27}^{36}$ ticket229.2. $_{27}^{36}$	fully solve the optimization problems with nonblocking calls. This is the only Fortran
57	support method that is consistent with the Fortran standard (Fortran 2008 $+$ TR
38	29113 and later). This method is highly recommended for all MPI applications.
ticket230-B. ³⁹	
ticket230-B. $^{40}_{41}$	2. [Extended Fortran Support]USE mpi: [An implementation with this level of
10	Fortran support provides Basic Fortran Support plus additional features that specifi-
ticket230-B.	cally support Fortran 90, as]This method is described in Section 16.2.3 and requires
ticket229.2. $^{43}_{44}$	compile-time argument checking. Handles are defined as INTEGER. This Fortran sup-
45	port method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
ticket 230-B. $_{46}$	recommended. It exists only for backwards compatibility.
ticket230-B. 47	3. [Basic Fortran Support]INCLUDE 'mpif.h': [An implementation with this level
48	of Fortran support provides the original Fortran bindings specified in $MPI-1$, with small

Unofficial Draft for Comment Only

ticket 233-E.

additional requirements specified] This method is described in Section 16.2.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

[A compliant MPI-2]Compliant MPI-3 implementations providing a Fortran interface must provide[Extended Fortran Support unless the target compiler does not support modules or KIND-parameterized types] [all three Fortran support methods.]provide one or both of the following:

- The USE mpi_f08 Fortran support method.
- The USE mpi and INCLUDE 'mpif.h' Fortran support methods.

Section 16.2.6 on page 656 describes restrictions if the compiler does not support all the needed features.

Application[s] subroutines and functions may use either [the mpi]one of the modules or the mpif.h include file. An implementation may require the use of one of the modules to prevent type mismatch errors[(see below)].

Advice to users. [It is recommended to use the mpi module even if it is not necessary to use it to avoid type mismatch errors]Users are advised to utilize one of the MPI modules even if mpif.h enforces type checking on a particular system. Using a module provides several potential advantages over using an include file; the mpi_f08 module offers the most [advantages]robust and complete Fortran support. (End of advice to users.)

[It]In a single application, it must be possible to link together routines [some of which USE mpi and others of which INCLUDE 'mpif.h'.]which USE mpi_f08, USE mpi, and INCLUDE mpif.h.

The INTEGER compile-time constant MPI_SUBARRAYS_SUPPORTED is set to .TRUE. if all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-rank [41]; otherwise it is set to .FALSE.. The INTEGER compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was added to the choice buffer arguments of all nonblocking interfaces **and** the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TR 29113), otherwise it is set to .FALSE.. These constants exist with each Fortran support method, but not in the C/C++ header files. The values may be different for each Fortran support method. All other constants and the integer values of handles must be the same for each Fortran support method.

Section 16.2.2 through 16.2.4 define the Fortran support methods. The Fortran interfaces of each MPI routine are shorthands. Section 16.2.5 defines the corresponding full interface specification together with the used linker names and implications for the profiling interface. Section 16.2.6 the implementation of the MPI routines for different versions of the Fortran standard. Section 16.2.7 summarizes major requirements for valid MPI-3.0 implementations with Fortran support. Section 16.2.8 and Section 16.2.9 describe additional functionality that is part of the Fortran support. MPI_F_SYNC_REG is needed for one of the methods to prevent register optimization problems. A[new] set

Unofficial Draft for Comment Only

1 $\mathbf{2}$ 3 ⁴ ticket229.2. 56 ⁷ ticket230-B. ⁸ ticket230-B. 9 ticket230-B. ¹⁰ ticket230-B. 11 ticket229.2. 12 13 14 ¹⁵ ticket247-S. 16 17 18 ticket230-B. 19 ticket230-B. ₂₀ ticket230-B. ₂₁ ticket230-B. $_{22}$ ticket230-B. ticket230-B. 23 ticket230-B. ²⁴ ticket230-B. 25 ticket230-B. 26 ticket 229.2. 27 ticket230-B. 28 ticket230-B. 29³⁰ ticket234-F. 31 32 ³³ ticket238-J. ³⁴ ticket229.1. 3536 ³⁷ ticket234-F. ³⁸ ticket238-J. ³⁹ ticket 229.1. ⁴⁰ ticket230-B. ⁴¹ ticket247-S. 4243 44⁴⁵ ticket238-J. 4647 ⁴⁸ ticket230-B.

	1	of functions to provides additional support for Fortran intrinsic numeric types, includ-
	2 3	ing parameterized types: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and
ticket 250-V.	4	MPI_TYPE_CREATE_F90_COMPLEX. [Parameterized]In the context of MPI, parameter-
	5	ized types are Fortran intrinsic types which are specified using KIND type parameters.
ticket230-B.		[These routines are described in detail in Section 16.2.9.]Sections 16.2.10 through 16.2.19
	7 8	give an overview and details on known problems when using Fortran together with MPI;
	9	Section $16.2.20$ compares the Fortran problems with those in C.
1:1 4000 D		16.2.2. Eastern Connect Threads the set SOO Made la
ticket230-B. ticket230-B.		16.2.2 Fortran Support Through the mpi_f08 Module
UCKet250-D.	12	An MPI implementation providing a Fortran interface must provide a module named mpi_f08
	13	that can be used in a Fortran program. Section $16.2.6$ on page 656 describes restrictions if
ticket 247-S.	14	the compiler does not support all the needed features. Within all MPI function specifica-
	15	tions, the first of the set of two Fortran routine interface specifications is provided by this
ticket230-B.		module. This module must:
ticket230-B.	17	• Define all named MPI constants.
	18	• Denne an named WFT constants.
	19	• Declare MPI functions that return a value.
	20	Describe som light intenformer som dien to the Destruction intenformer medications
	21 22	• Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments
ticket241-M.		which are not TYPE(*), with the following exception:
ticket230-B.		which are not 111 L(*), with the following exception.
ticket229.1.		Only one Fortran interface is defined for functions that are deprecated as of
	26	MPI-3.0. This interface must be provided as an explicit interface according to
	27	the rules defined for the mpi module, see Section $16.2.3$ on page 646 .
	28	Advice to users. It is strongly recommended that developers substitute calls
	29	to deprecated routines when upgrading from mpif.h or the mpi module to
ticket231-C.	30	the mpi_f08 module. (End of advice to users.)
0101101201 01	31	
	32	• Define all MPI handles with uniquely named handle types (instead of INTEGER handles,
tieleet941 M	33	as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this decument (except for the depresented routines)
ticket241-M. ticket238-J.		function definition throughout this document (except for the deprecated routines).
UCKC0250-5.	36	• Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations,
ticket229.1.		and set the INTEGER compile-time constant $MPI_ASYNC_PROTECTS_NONBLOCKING$
	38	to .TRUE. if the underlying Fortran compiler supports the $\texttt{ASYNCHRONOUS}$ attribute
	39	for MPI communication (as part of TR 29113). See Section $16.2.6$ on page 656 for
ticket234-F.	40	older compiler versions.
	41	• Set the INTEGER compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. and
	42	declare choice buffers using the Fortran 2008 TR 29113 feature assumed-type and
ticket229.2.	43	assumed-rank, i.e., TYPE(*), DIMENSION() in all nonblocking, split collective and
	44	persistent communication routines, if the underlying Fortran compiler supports it.
ticket229.2.		With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.
	46	
	47	
	48	

ticket230-B. ticket230-B. Rationale. In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TR 29113 feature is not needed for the support of noncontiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TR 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 16.2.6 on page 656 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

Rationale. For these definitions in the mpi_f08 bindings, in most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT and INOUT dummy arguments that allow one of the non-ordinary Fortran constants (see MPI_BOTTOM, etc. in Section 2.5.4 on page 15) as input, an INTENT is not specified. (End of rationale.)

Advice to users. If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [39], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of INTENT(OUT) the copy-in may be suppressed by the optimization and the routine is starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 16.2.3 on page 646. (End of advice to users.)

• Declare all ierror output arguments as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and predefined callbacks (e.g., MPI_NULL_COPY_FN).

Rationale. For user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_NULL_COPY_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

Unofficial Draft for Comment Only

 $\mathbf{2}$

1

ticket230-B.

ticket229.1.

1 2	The MPI Fortran bindings in the mpi_f08 module are designed based on the Fortran
3 4	2008 standard [39] together with the Technical Report "TR 29113 Further Interoperability with C" [41] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.
5	Rationale. The features in TR 29113 on further interoperability with C were decided
6 7	on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a
8	higher level of integration between Fortran-specific features and C than was provided
9	in the Fortran 2008 standard; part of this design is based on requirements from the
ticket229.1. 10	MPI Forum to support MPI-3.0. According to [.][40] page iv, last paragraph, "it is
11	the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified
12	by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this
13	feature identifies errors that need to be corrected, or changes are needed to achieve
14 15	proper integration, in which case every reasonable effort will be made to minimize the
ticket229.1. $_{16}$	impact of such changes on existing implementations".
17	The TR 29113 contains the following language features that are needed for the MPI
18	bindings in the mpi_f08 module: assumed-type and assumed-rank. It is important
19	that any possible actual argument can be used for such dummy arguments, e.g.,
20	scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and
21 22	with any element type, e.g., REAL, CHARACTER*5, CHARACTER*(*), sequence derived
22	types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation
24	of a choice buffer dummy argument (e.g., with mpif.h without argument-checking)
25	can be used in an implementation with assumed-type and assumed-rank argument in
26	an explicit interface (e.g., with the mpi_f08 module).
27	The INTERFACE construct in combination with BIND(C) allows the implementation of
28	the Fortran mpi_f08 interface with a single set of portable wrapper routines written
29 30	in C, which supports all desired features in the $\texttt{mpi_f08}$ interface. TR 29113 also has
31	a provision for OPTIONAL arguments in BIND(C) interfaces.
32	A further feature useful for MPI is the extension of the semantics of the
33	ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to
34	protect buffers of Fortran asynchronous I/O. With TR29113, this attribute now also
35	covers asynchronous communication occurring within library routines written in C.
36 37	The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (<i>End of rationale.</i>)
38	1 122.3 and web committee. (Ena of fationale.)
39	
ticket 230-B. 40	16.2.3 [Extended]Fortran Support Through the mpi Module
ticket230-B. ⁴¹ ticket230-B. ⁴²	[Implementations with Extended Fortran support must provide:
43	1. An mpi module
44	
45	2. (List item was moved to another location, and modified) A new set of func-
46 47	tions to provide additional support for Fortran intrinsic numeric types, including parameterized types: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE,
48	MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and
	Unofficial Draft for Comment Only

MPI_TYPE_CREATE_F90_COMPLEX. Parameterized types are Fortran intrinsic types which are specified using KIND type parameters. These routines are described in detail in Section 16.2.9.	1 2 3
Additionally, high-quality implementations should provide a mechanism to prevent fatal type mismatch errors for MPI routines with choice arguments.	4 5 6
The mpi Module	7 8 9
An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran [90] program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:	¹⁰ ticket230-B. ¹¹ ticket230-B. ¹² ¹³
• Define all named MPI constants	15
• Declare MPI functions that return a value.	$^{16}_{17}$ ticket230-B.
• Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists.	18 19 20 21
• Define all MPI handles as type INTEGER.	²² ticket231-C.
• Define all named handle types and the derived type MPI_Status that are used in the mpi_f08 module.	$^{23}_{24}$ ticket243-O. $^{25}_{25}$ ticket231-C. $^{26}_{26}$ ticket231-C.
<i>Rationale.</i> They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (<i>End of rationale.</i>)	²⁷ ²⁸ ²⁹ ticket238-J.
• A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi_f08 module if it is supported by the underlying compiler.	30 31 32 33
• Set the INTEGER compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces and the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TR 29113), otherwise to .FALSE	³⁴ ticket229.1. ³⁶ ³⁷ ³⁸ ticket247-S.
Advice to users. For an MPI implementation that fully supports nonblocking calls with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses 'contiguous' but not 'simply contiguous' ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constaints for ASYNCHRONOUS dummy arguments. This is not considered	40 41 42 ⁴³ ticket229.1. ⁴⁴ ticket229.1. 45 46 47

Unofficial Draft for Comment Only

a violation of backward compatibility because existing applications can not use the

ASYNCHRONOUS attribute to protect nonblocking calls. Onother reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copyin/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 16.2.12 on page 673 for more details. (*End of advice to users.*)

- A high quality MPI implementation may enhance the interface by using TYPE(*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !\$PRAGMA IGNORE_TKR or a set of overloaded functions as described by M. Hennecke in [28], if the compiler supports this TR 29113 language feature. See Section 16.2.6 on page 656 for further details.
- Set the INTEGER compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(*), DIMENSION(..), otherwise set it to .FALSE.. With MPI_SUBARRAYS_SUPPORTED==.TRUE., non-contiguous subarrays can be used as buffers in nonblocking routines.
- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TR 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be disallowed. See Section 16.2.6 on page 656 for details.

An MPI implementation may provide other features in the mpi module [other features] that enhance the usability of MPI while maintaining adherence to the standard. For example, it may[:] provide INTENT information in these interface blocks. [

- Provide interfaces for all or for a subset of MPI routines.
- Provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI [generic interface]language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi_f08 bindings. (End of advice to implementors.)

Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran INTENT. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI_BOTTOM with a dummy OUT argument. Moreover, "constants" such as MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran, but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent [is]was changed in several places [by]in MPI-2. For instance, MPI_IN_PLACE changes the [sense]intent of an OUT argument to be INOUT. (*End of rationale.*)

	1
	2
	3
	4
ticket 229.2 .	5
	6
${\rm ticket 232\text{-}D}.$	7
	8
	9
	10
	11
	12
	13
	14
ticket229.2.	15 16
	10
	18
	19
	20
	20
	22
	23
	24
	25
ticket230-B.	26
ticket230-B.	27
ticket230-B.	28
ticket230-B.	29
	30
	31
	32
	33
	34
	35
ticket 230-B.	
ticket242-N.	
	38
	39
	40
	41
	42
	43
	44 45
ticket230-B.	45 46
	40
ticket250-V.	

ticket242-N.

1 Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 $\mathbf{2}$ that "INTENT(OUT) means that the value of the argument after invoking the pro-3 cedure is entirely the result of executing that procedure. If an argument should retain 4 its value rather than being redefined, INTENT(INOUT) should be used rather than $\mathbf{5}$ INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-6 $\overline{7}$ TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is *definable*". Applications that include mpif.h may not expect that 8 9 INTENT (OUT) is used. In particular, output array arguments are expected to keep their content as long as the MPI routine does not modify them. To keep this behavior, it is 10 11 recommended that implementations not use INTENT(OUT) in the mpi module and the mpif.h include file, even though INTENT(OUT) is specified in an interface description of the mpi_f08 module. (End of advice to implementors.)

[(Paragraph was moved to another location, and modified) Applications may use either the mpi module or the mpif.h include file. An implementation may require use of the module to prevent type mismatch errors (see below).

Advice to users. It is recommended to use the mpi module even if it is not necessary to use it to avoid type mismatch errors on a particular system. Using a module provides several potential advantages over using an include file. (*End of advice to users.*)

It must be possible to link together routines some of which USE mpi and others of which INCLUDE mpif.h.]

No Type Mismatch Problems for Subroutines with Choice Arguments

A high-quality MPI implementation should provide a mechanism to ensure that MPI choice arguments do not cause fatal compile-time or run-time errors due to type mismatch. An MPI implementation may require applications to use the **mpi** module, or require that it be compiled with a particular compiler flag, in order to avoid type mismatch problems.

Advice to implementors. In the case where the compiler does not generate errors, nothing needs to be done to the existing interface. In the case where the compiler may generate errors, a set of overloaded functions may be used. See the paper of M. Hennecke [28]. Even if the compiler does not generate errors, explicit interfaces for all routines would be useful for detecting errors in the argument list. Also, explicit interfaces which give INTENT information can reduce the amount of copying for BUF(*) arguments. (End of advice to implementors.)

]

16.2.4 [Basic]Fortran Support Through the mpif.h Include File

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

[Because Fortran 90 is (for all practical purposes) a superset of Fortran 77, Fortran 90 (and future) programs can use the original Fortran interface.][The following additional requirements are added:] An MPI implementation providing a Fortran interface must pro-

Unofficial Draft for Comment Only

¹² ¹³ ¹⁴ ticket230-B.

15

16

17 18

19

20

25 26

27

28

29

30

31 32

33

34

35

²³ ticket232-D.

 $^{42}_{43}$ ticket230-B.

44 ticket230-B.

- ticket233-E.
- $^{43}_{46}$ ticket230-B.

⁴⁷ ticket230-B. ⁴⁸ ticket230-B.

		650 CHAPTER 16. LANGUAGE BINDINGS
ticket230-B.	4	vide an include file named mpif.h that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:
	5 6 7	1. Implementations are required to provide the file mpif.h, as described in the original MPI-1 specification.
	8 9 10	2. mpif.h must be valid and equivalent for both fixed- and free- source form.
	11 12	• Define all named MPI constants.
	13 14	• Declare MPI functions that return a value.
	15	• Define all handles as INTEGER.
	16 17	• Be valid and equivalent for both fixed and free source form.
	18 19 20	For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted).
ticket229.1.	21 22 23 24 25	• Set the INTEGER compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the mpi module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to .FALSE
	26 27	Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is strongly encouraged for the following reasons:
	28 29 30 31	 Most mpif.h implementations do not include compile-time argument checking. Therefore, too many bugs in MPI applications remain undetected at compile-time, such as:
	32 33 34	 Missing ierror as last argument in most Fortran bindings. Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.
	35 36 37 38	 Wrong argument positions; e.g., interchanging the count and datatype arguments. Passing wrong MPI handles; e.g., passing a datatype instead of a communi-
	39 40 41 42	 cator. The migration from mpif.h to the mpi module should be relatively straightforward (i.e., substituting include 'mpif.h' after an implicit statement by use mpi before such implicit statement) as long as the application syntax is correct.
	43 44 45 46	• Migrating portable and correctly written applications to the mpi module is not expected to be difficult. No compile or runtime problems should occur because an mpif.h include file was always allowed to provide explicit Fortran interfaces.
	40 47 48	(End of advice to users.)

Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to retain strong backward compatibility. Internally, mpif.h and the mpi module may be implemented so that the same (or similar) library implementation of the MPI routines can be used. (*End of rationale.*)

Advice to implementors. To make mpif.h compatible with both fixed- and freesource forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-form line length, it is recommended that [requirement two]the requirement of usability in free and fixed source form applications be met by constructing mpif.h without any continuation lines. This should be possible because mpif.h [contains]may contain only declarations, and because common block declarations can be split among several lines. The argument names may need to be shortened to keep the SUBROUTINE statement within the allowed 72-6=66 characters, e.g.,

INTERFACE
SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
 ... ! dummy argument declarations

This line has [65]65 characters and is the longest in MPI-3.0.

TODO: This is only checked for MPI-2.2. We have to check all new MPI-3.0 interfaces that they stay within these 66 characters. Otherwise the routine name should be shortened before the name is standardized.

[If mpif.h contains also explicit interfaces with BIND(C,NAME='...') for providing MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING equals .TRUE., the linker routine name may need to be shortened.]As long as the MPI standard contains routines with choice buffers and a name length and argument count that implies that a BIND(C) implementation would need to shorten their linker names in mpif.h, the mpif.h cannot set MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING equals .TRUE., because such shortening is invalid. For example, MPI_FILE_WRITE_AT_ALL_BEGIN with 6 arguments, may be defined:

INTERFACE MPI_FILE_WRITE_AT_ALL_BEGIN
SUBROUTINE MPI_X(a,b,c,d,e,f)BIND(C,NAME='MPI_File_write_at_all_begin_f')
... ! dummy argument declarations

This would need a line length of 73 characters, i.e., the C routine name [must]would need to be shortened by 7 characters to stay within the available 66 characters. [TODO: Do we want to define these shortened routine names for mpif.h; this would help the tools people.]Note that the name MPI_X has no meaning for the compilation, and that this problem occurs only with routines with choice buffers implemented with the assumed-type and assumed-rank facility of TR 29113. To support Fortran 77 as well as Fortran 90 and later, it may be necessary to eliminate all comments from mpif.h. (End of advice to implementors.)

16.2.5 Interface Specifications, Linker Names and the Profiling Interface

The Fortran interface specifications of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments

Unofficial Draft for Comment Only

```
\mathbf{2}
3
4
5
6
\overline{7}
8
9 ticket230-B.
10
11 ticket230-B.
12
13 ticket230-B.
14 ticket229.1.
15
16
17
18
   ticket 229.4.
19
   ticket 229.1
20
21
22 ticket229.1.
23
^{24} ticket 229.1.
25
26
27
^{28} ticket 229.1.
29
30
^{31}
32
33
34
35
36 ticket 229.1.
37
38 ticket229.4.
39
40
41
42 ticket230-B.
43
44
<sup>45</sup> ticket247-S.
46 ticket247-S.
47
48
```

1 2 3 4 5	together with additional attributes. The rules for the linker names and its implications for the profiling interface are specified within this section. The linker name of a Fortran routine is defined as the name that a C routine would have if both routines would have the same name visible for the linker. A typical linker name of the Fortran routine FOOfoo is
6	foofoo In the case of BIND(C,NAME=''), the linker name is directly defined through the given string.
7	The following rules for linker names apply:
8	
9	• With the Fortran mpi_f08 module, if MPI_SUBARRAYS_SUPPORTED equals .TRUE.:
10 11 12	The Fortran binding must use BIND(C) interfaces with an interface name identical to the language independent name, e.g., MPI_SEND. The linker name is a combination of the C name and an _f08 suffix, e.g., MPI_Send_f08. Prototype example:
13	INTERFACE
14	SUBROUTINE MPI_Send() BIND(C,NAME='MPI_Send_f08')
15	• With the Fortran mpi_f08 module, if MPI_SUBARRAYS_SUPPORTED equals .FALSE.
16	(i.e., with a preliminary implementation of this module without TR 29113):
17 18	The linker name of each routine is defined through the linker name mapping of the
19	Fortran compiler for the name defined when subarrays are supported. For example,
20	MPI_Send_f08 may be mapped to mpi_send_f08 Example:
21	INTERFACE MPI_Send
22	SUBROUTINE MPI_Send_f08()
23	• With the Fortran mpi module or mpif.h include file, if MPI_SUBARRAYS_SUPPORTED
24	equals .FALSE.:
25 26	The linker name of each routine is defined through the linker-name mapping of the
27	Fortran compiler. For example, MPI_SEND may be mapped to mpi_send Example:
28	INTERFACE
29	SUBROUTINE MPI_SEND()
30	• With the Fortran mpi module or mpif.h include file, if MPI_SUBARRAYS_SUPPORTED
31 32	equals .TRUE.:
33	The Fortran binding must use BIND(C) interfaces with an interface name identical to
34	the language independent name, e.g., MPI_SEND. The linker name is a combination
35	of the C name and an _f suffix, e.g., MPI_Send_f. Prototype example:
36	INTERFACE
37	SUBROUTINE MPI_SEND() BIND(C,NAME='MPI_Send_f')
38	If the support of subarrays is different for the mpi module and the mpif.h include file,
39	then both linker-name methods can be used in the same application. If the application also
40	uses the mpi_f08 module and was compiled with this module partially before and after the
41 42	subarrays were supported, then all four interfaces are used within the same application.
42	
44	Rationale. After a compiler provides the facilities from TR29113, i.e., TYPE(*), DIMENSION(), it is possible to change the bindings within a Fortran support method
45	to support subarrays and without recompiling the complete application. Of course,
46	only recompiled routines can benefit from the added facilities. There is no binary com-
47	patibility conflict because each interface uses its own linker names and all interfaces
48	use the same constants and type definitions. (End of rationale.)

Unofficial Draft for Comment Only

A user-written or middleware profiling routine that is written according to the same binding rules will have the same linker name, and therefore, can interpose itself as the MPI library routine. The profiling routine can internally call the matching PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments. In this case, the profiling software must use the same Fortran support method as used in the calling application program, because the C, mpi_f08 and mpi callback prototypes are different.

Advice to users. This advice is mainly for tool writers. Even if an MPI library supports subarrays in all three Fortran support methods, a portable profiling layer should also provide the two interfaces for MPI_SUBARRAYS_SUPPORTED==.FALSE. to support older binary user routines that were compiled before TR29113 level support was achieved.

If a user application calls MPI_SEND, then the chosen Fortran support method together with the MPI implement decision about MPI_SUBARRAYS_SUPPORTED imply, to which linker name the compiler will translate this call, i.e., whether the application calls mpi_send__, or MPI_Send_f, or mpi_send_f08__, or MPI_Send_f08. If the profiling layer wants to be independent of the decision of the user program and MPI implementation, then it should provide all four routines. For example:

SUBROUTINE MPI_SEND()	<pre>BIND(C,NAME='MPI_Send_f')</pre>
USE mpi	
CALL PMPI_SEND()	
END SUBROUTINE	

The MPI library must provide the PMPI_SEND routine according to the same rules as for providing the MPI_SEND routine. (*End of advice to users.*)

Advice to implementors. If an implementation provides in a first step two sets of routines, one for the mpi module and mpif.h, and the other for the mpi_f08 module, and both sets without TR 29113, i.e., MPI_SUBARRAYS_SUPPORTED equals .FALSE., [If] and the implementor wants to add a TR 29113 based set of routines, then it is not necessary to add two full sets of routines. For full quality, it is enough to implement in each set only those routines that have a choice buffer argument. (End of advice to implementors.)

In the case that a Fortran binding consists of multiple routines through function overloading, the base names of overloaded routines are appended by a suffix notifying the difference in the argument list. For example, MPI_ALLOC_MEM (in the mpi module and mpif.h) has an INTEGER(KIND=...) baseptr argument without a suffix. This routine is overloaded by a routine with TYPE(C_PTR) baseptr and the suffix _CPTR. The implied linker name base is MPI_ALLOC_MEM_CPTR. It is mapped to the linker names MPI_Alloc_mem_cptr_f, and, e.g., mpi_alloc_mem_cptr__. Note that these routines are always called via the interface name MPI_ALLOC_MEM by the application within all Fortran support methods.

For routines without ASYNCHRONOUS choice buffers and that are not predefined callback routines, the implementor can freely choose to implement the routines according to the rules for MPI_SUBARRAYS_SUPPORTED equals .TRUE. or .FALSE., provided that the following rule about routine grouping is fulfilled. The implementation of routines with ASYNCHRONOUS choice buffers depends on the rules for the provided Fortran support method and language

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

12

13 14

15

16

17

18

19

25

26 27

28

29

30

32

33

34 35

36

37

38

39

40

41

42

44

45

46

47

48

43 ticket 229.2.

31 ticket229.2.

level of the underlying compiler. Predefined callback routines for the mpi_f08 module
 must be implemented with BIND(C) interfaces, and for the mpi module and mpif.h without
 BIND(C).

Similar MPI routines are grouped together for linker symbol scheme classification. If
 the peer routine of a group is available within an MPI library with one of its possible linker
 names then all of the routines in this group must provided according to the same linker
 name scheme. If the peer routine is not available through a linker name scheme then all
 other routines in the group nust not be available through this scheme.

⁹ Peer routines and their groups:

	0h	
10 11	MPI_ALLOC_MEM	MPI_ALLOC_MEM and MPI_WIN_ALLOCATE.
12	MPI_FREE_MEM	Only this routine is in this group.
13	MPI_GET_ADDRESS	MPI_GET_ADDRESS and MPI_ADDRESS.
	MPI_SEND	All routines with choice buffer arguments that
14	_	are not declared as ASYNCHRONOUS within the
15		mpi_f08 module.
16		-
17	MPI_ISEND	All routines with choice buffer arguments
18		that are declared as ASYNCHRONOUS within the
19		mpi_f08 module.
20	MPI_OP_CREATE	Only this routine is in this group.
20	MPI_REGISTER_DATAREP	Only this routine is in this group.
	MPI_COMM_KEYVAL_CREATE	All other routines with callback function argu-
22		ments.
23	MPI_COMM_DUP_FN	All predefined callback routines.
24		•
25	MPI_COMM_RANK	All other MPI routines.

Additionally, four C preprocessor macros are available in mpi.h for each routine group. The name of the macros are the peer routine name written as in the list above and appended with one of the following suffixes and meanings:

mai f00 PIND C	The macro is set to 1 if the BIND(C) linker name with the
	linker suffix _f08 is available for all routines within this group
	(e.g., MPI_Send_f08), otherwise it is set to 0.
_mpi_f08_BIND_F	The macro is set to 1 if the Fortran linker name with the
	linker suffix _f08 is available for all routines within this group
	(e.g., $mpi_send_f08_$), otherwise it is set to 0.
_mpi_BIND_C	The macro is set to 1 if the BIND(C) linker name with the
	linker suffix _f is available for all routines within this group
	(e.g., MPI_Send_f), otherwise it is set to 0.
mpi BIND E	The macro is set to 1 if the Fortran linker name without
	a linker suffix is available for all routines within this group
	(e.g., $mpi_send_$), otherwise it is set to 0.
For example	
	_mpi_f08_BIND_F

44 ...
45 #define MPI_SEND_mpi_f08_BIND_C 0
46 #define MPI_SEND_mpi_f08_BIND_F 1
47 #define MPI_SEND_mpi_BIND_C 0
48 #define MPI_SEND_mpi_BIND_F 1

26

27

<pre>#define MPI_ISEND_mpi_f08_BIND_C</pre>	1 2
#define MPI_ISEND_mpi_f08_BIND_F	1 ³
#define MPI_ISEND_mpi_BIND_C #define MPI_ISEND_mpi_BIND_F	1 4 1 5
"define in i_ibbnb_mpi_binb_i	6
#define MPI_COMM_DUP_FN_mpi_f08_B	IND_C 1 7
#define MPI_COMM_DUP_FN_mpi_f08_B	
<pre>#define MPI_COMM_DUP_FN_mpi_BIND_</pre>	C 0 9
<pre>#define MPI_COMM_DUP_FN_mpi_BIND_</pre>	
	1:
shows, that	1:
• the routines in the MPI_SEND group ar names (e.g., mpi_send_f08, mpi_send_	re only available through their Fortran linker _, mpi_recv_f08, mpi_recv,),
library, the mpi_f08 and mpi modules (
For the predefined callbacks, there is no che callback function prototypes which are BIND(for the mpi module and mpif.h.	(C) based for mpi_f08 and without BIND(C) 23
Advice to implementors. If all following most compilers):	; conditions are fulfilled (which is the case for 2^{24}
• the handles in the mpi_f08 module (same as an INTEGER handle),	e occupy one Fortran numerical storage unit
	l to pass an actual ierror argument to a non binary compatible to passing an actual ierror ment that is declared as OPTIONAL,
• the internal argument passing for ments is the same,	ASYNCHRONOUS and non-ASYNCHRONOUS argu- 34
• the internal routine call mechanism piler,	The same for the Fortran and the C com- 30
• the compiler does not provide TR 2	29113, ³⁴
tations for all Fortran support methods	 may use the same internal routine implemen- but with several different linker names. For ded only for the routine group of MPI_ISEND. / _mpi_f08_BIND_F / _mpi_BIND_C / 44

1 2 3 4 5 6 7 8 9 10 11 11 12 13	MPI_ALLOC_MEM MPI_FREE_MEM MPI_GET_ADDRESS MPI_SEND MPI_ISEND MPI_OP_CREATE MPI_REGISTER_DATAREP MPI_COMM_KEYVAL_CREATE MPI_COMM_DUP_FN MPI_COMM_RANK (End of advice to implementors.)	Without TR 29113 0/1/0/1 0/1/0/1 0/1/0/1 0/1/0/1 0/1/0/1 0/1/0/1 1/0/1 1/0/1 0/1/0/1	Upgrade to TR 29113 0/1/0/1 0/1/0/1 0/1/0/1 1/1/1/1 0/1/0/1 0/1/0/1 0/1/0/1 1/0/0/1 0/1/0/1	Upgrade for strided data optimization 0/1/0/1 0/1/0/1 1/1/1/1 1/1/1/1 0/1/0/1 0/1/0/1 0/1/0/1 1/0/0/1 0/1/0/1 0/1/0/1	New impl. with TR 29113 1/0/1/0 1/0/1/0 1/0/1/0 1/0/1/0 1/0/1/0 1/0/1/0 1/0/1/0 1/0/1/0 1/0/1/1 1/0/1/0	
14 15						
ticket 247-S. ¹⁶	16.2.6 MPI for Different Fortran Stan	dard Versio	ns			
ticket247-S. 17 18 19	This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.					
20	• For Fortran 77 with some extensions:					
21 22	- MPI identifiers are limited to thirty or more, not six, significant characters.					
23	 MPI identifiers may contain underscores after the first character. An MPI subroutine with a choice argument may be called with different argument types. 					
24						
25 26						
- Although not required b the MPI standard, the INC						
28	available for including mpif.h into the user application source code.					
29 30	Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute ad- dresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address					
31	does not fit into the memory space		•	· · · · · · · · · · · · · · · · · · ·		
32	is solved with MPI_GET_ADDRES			No. 1	1	
33 34	• For Fortran 90:					
The major additional features that are needed from F		l from Fortra	tran 90 are:			
³⁶ – The MODULE and INTERFACE concept.						
37	- The KIND= and SELECTEDKIND concept.					
38 39	 Fortran derived TYPEs and the SEQUENCE attribute. 					
40	- The OPTIONAL attribute for d					
41				eded for the		
42 43	use of MPI_ALLOC_MEM.					
44 45	With these features, MPI-1.1 - MPI-2.2 can be implemented without restrictions. MPI-					
45	3.0 can be implemented with some restrictions. The Fortran support methods are abbreviated with $S1 = \text{the mpi}_{100} \text{module}$, $S2 = \text{the mpi} \text{module}$, and $S3 = \text{the}$					
47 48	mpif.f include file. If not stated of prevent implementing the complet	otherwise, r	estrictions ex			

•

.

 MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non- contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
 S1, S2, and S3 can be implemented, but for S1, only a preliminary implementation is possible.
- In this preliminary interface of $S1$, the following changes are necessary:
 * The routines are not BIND(C). * TYPE(*), DIMENSION() is substituted by non-standardized extensions
like !\$PRAGMA IGNORE_TKR. 10
* The ASTNCHRUNUUS attribute is omitted.
* PROCEDURE() callback declarations are substituted by EXTERNAL.
- The linker names are specified in Section $16.2.5$ on page 651 .
 Due to the rules specified in Section 16.2.5 on page 651, choice buffer declarations should be implemented only with non-standardized extensions like !\$PRAGMA IGNORE_TKR (as long as F2008+TR29113 is not available).
In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 16.2.11 on 20
page 672). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 16.2.15 on page 677.
Only in S1: The implicit interfaces for routines with choice buffer arguments ²³
imply that the ierror argument cannot be defined as OPTIONAL. For this reason, ²⁴
it is recommended not to provide the mpi_f08 module if such an extension is not available.
 The ASYNCHRONOUS attribute can not be used in applications to protect buffers in nonblocking MPI calls (S1-S3).
 The TYPE (C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines is not available.
 In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types ralated to TYPE(MPI_Status) (see Section 16.3.5 on page 698) are also not available in the mpi module and mpif.h.
For Fortran 95:43The quality of the MPI interface and the restrictions are the same as with Fortran 90.44
45
For Fortran 2003:46The major features that are needed from Fortran 2003 are:47
- Interoperability with C, i.e., ⁴⁸

1	* BIND(C, NAME='') interfaces.
2	* BIND(C) derived types.
3	* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.
4	
5	 The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.
6	
7	- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O.
8 9	This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TR 29113.
9 10	MET communication in the TR 29115.
11	With these features (but still without the features of TR29113), MPI-1.1 - MPI-2.2 $$
12	can be implemented without restrictions, but with one enhancement:
13	- The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as
14	long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR
15	and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a void
16	* argument.
17	
18	MPI-3.0 can be implemented with the following restrictions:
19 20	- MPI_SUBARRAYS_SUPPORTED equals .FALSE
21	- For S1, only a preliminary implementation is possible. The following changes are
22	necessary:
23	* The routines are not BIND(C).
24	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
25	like !\$PRAGMA IGNORE_TKR.
26 27	- The linker names are specified in Section 16.2.5 on page 651.
28	- With S1, the ASYNCHRONOUS is required as specified in the second Fortran inter-
29	faces. With S2 and S3 the implementation can also add this attribute if explicit
30	interfaces are used.
31	- The ASYNCHRONOUS Fortran attribute can be used in applications to try to pro-
32	tect buffers in nonblocking MPI calls, but the protection can work only if the
33	compiler is able to protect asynchronous Fortran I/O and makes no difference
34	between such a synchronous Fortran I/O and MPI communication.
35 36	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE
30	routines can be used only for Fortran types that are C compatible.
38	– The same restriction as for Fortran 90 applies if non-standardized extensions like
39	!\$PRAGMA IGNORE_TKR are not available.
40	
41	• For Fortran $2008 + TR 29113$ and later and For Fortran $2003 + TR 29113$:
42	The major feature that are needed from TR29113 are:
43	The major reasure that are needed from 11(2911) are.
44	- TYPE(*), DIMENSION() is available.
45	- The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com-
46 47	munication.
47	- OPTIONAL dummy arguments are allowed in combination with BIND(C) interfaces.

- CHARACTER(LEN=*) dummy arguments are allowed in combination with BIND(C)	1
interfaces.	3
$-$ The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not	4
restricted to Fortran types for which a corresponding type in C exists.	5
Using these features, MPI-3.0 can be implemented without any restrictions.	6
Using these reatures, wit i-5.0 can be implemented without any restrictions.	7
- With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS at-	8
tribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR)	9
binding of the MPI_ALLOC_MEM and $MPI_WIN_ALLOCATE$ routines can be	9 10
used for any Fortran type.	10
$-$ With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation	11
dependent. A high quality implementation will also provide	12
MPI_SUBARRAYS_SUPPORTED == . TRUE. and will use the	13
ASYNCHRONOUS attribute in the same way as in S1.	14
 If non-standardized extensions like !\$PRAGMA_IGNORE_TKR are not available then 	15
S2 must be implemented with TYPE(*), DIMENSION().	17
52 must be implemented with TIPE(*), DIMENSION().	$_{18}^{17}$ ticket234-F.
Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice	19
argument may be implemented with an explicit interface using compiler directives,	20
for example:	20
	21
INTERFACE	23
SUBROUTINE MPI(buf,)	23
!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf	24
!\$PRAGMA IGNORE_TKR buf	26
!DIR\$ IGNORE_TKR buf	
!IBM* IGNORE_TKR buf	27
REAL, DIMENSION(*) :: buf	28
! declarations of the other arguments	29
END SUBROUTINE	30
END INTERFACE	31
(End of advice to implementors.)	32
	33
	34
7 Requirements on Fortran Compilers	$_{36}^{35}$ ticket230-B.

16.2.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 16.2.11 through 16.2.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi_f08 and mpi modules, and mpif.h) are:

Unofficial Draft for Comment Only

ticket230-B.

37

3839

40

41

42

43

44

4546

47

⁴⁸ ticket234-F.

1 2 ticket229.1. 3 4	• The language features assumed-type and assumed-rank from Fortran 2008 TR 29113 [41] are available. This is required only for mpi_f08. As long as this requirement is not supported by the compiler, it is valid to build [a preliminary MPI-3.0 (and not later)]an MPI library that implements the mpi_f08 module with MPI_SUBARRAYS_SUPPORTED
ticket230-B. 5 ticket229.1. 6 7 8 9	 set to .FALSE 'Simply contiguous' arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 16.2.12 on
ticket237-I. 10 11 12 13 ticket232-D. ¹⁴	 page 673 for more details. SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE
ticket234-F. ¹⁵ 16 17 18 ticket246-Q. 19	• All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., CHARACTER(LEN=*) strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
²⁰ 21 ticket238-J. ²² 23	• The array dummy argument of the ISO_C_BINDING intrinsic module procedure C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.
²³ 24 25 ticket238-J. ²⁶	• The Fortran compiler shall not provide TYPE(*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TR 29113. Specifically, the TR 29113 must be implemented as a whole.
27 28 ticket229.1. ²⁹ 30 ticket238-J. ³¹	The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TR 29113 and there is still one Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==.FALSE It is helpful when these rules are observed, especially for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows:
32 33 34 ticket238-J. 36	• Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI_F_SYNC_REG on page 684 and Section 16.2.8 on page 661, and DD on page 685) solve the problems described in Section 16.2.17 on page 680.
37 38 39 ticket238-J. 40 41	• The problems with temporary data movement (described in detail in Section 16.2.18 on page 687) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective IO) and the computation when overlapping communication and computation.
42 43 44 ticket230-B. 45	• Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 16.2.19 on page 688) are resolved without any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in calling MPI operations.
46 47 48	All of these rules are valid independently of whether the MPI routine interfaces in the mpi_f08 and mpi modules are internally defined with an INTERFACE or CONTAINS construct, and with or without BIND(C), and also when mpif.h uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard if the MPI interfaces are defined without BIND(C). Additional compiler support may be necessary if BIND(C) is used. Some of these additional requirements are defined in the Fortran 2008 TR 29113 [41]. Some of these requirements for MPI-3.0 are beyond the scope of TR 29113. (*End of advice to implementors.*)

Further requirements apply when the MPI library internally uses BIND(C) routine interfaces (i.e, for a full implementation of mpi_f08):

- Non-buffer arguments are INTEGER, INTEGER(KIND=...), CHARACTER(LEN=*), LOGICAL, and BIND(C) derived types, (handles and status in mpi_f08) variables and arrays; function results are DOUBLE PRECISION. All these types must be valid as dummy arguments in the BIND(C) MPI routine interfaces. When compiling an MPI application, the compiler should not issue warnings indicating that these types may not be interoperable with an existing type in C. Some of these types are already valid in BIND(C) interfaces since Fortran 2003, some may be valid based on TR 29113 (e.g., CHARACTER*(*)).
- OPTIONAL dummy arguments are also valid within BIND(C) interfaces. This requirement is fulfilled if TR 29113 is fully supported by the compiler.

16.2.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 16.2.17 on page 680, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.

MPI_F_SYNC_REG(buf) INOUT buf	initial address of buffer (choice)
<pre>MPI_F_sync_reg(buf) BIND(C) TYPE(*), DIMENSION(),</pre>	ASYNCHRONOUS :: buf
<pre>MPI_F_SYNC_REG(buf)</pre>	

This routine is a no-operation. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

Rationale. This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

		002 CHAI TER 10. LANGUAGE DINDINGS
	1 2 3 4 5	explicitly allowed to implement this routine within the mpi_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (<i>End of advice to implementors.</i>)
ticket229.1.	8 9	Rationale. This routine is not defined with TYPE(*), DIMENSION(*), i.e., assumed size instead of assumed rank, because this would restrict the usability to 'simply contiguous' arrays and would require overloading with another interface for scalar arguments. (<i>End of rationale.</i>)
ticket229.1.	10 11 12 13 14 15 16 17	Advice to users. If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI_F_SYNC_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not to be called if MPI_ASYNC_PROTECTS_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (End of advice to users.)
ticket230-B.	18	16.2.9 Additional Support for Fortran Numeric Intrinsic Types
UCKet250-B.	20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36	[The routines in this section are part of Extended Fortran Support described in Sec- tion 16.2.3.] MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types. Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL and CHARACTER) with an optional integer KIND parameter that selects from among one or more variants. The specific meaning of different KIND values themselves are implementation dependent and not specified by the language. Fortran provides the KIND selection functions selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX and INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two declarations are equivalent:
	38 39 40	double precision x real(KIND(0.0d0)) x
ticket230-B.	41 42 43 44	MPI provides two orthogonal methods to communicate using numeric intrinsic types. The first method (see the following section) can be used when variables have been de- clared in a portable way — using default KIND or using KIND parameters obtained with the selected_int_kind or selected_real_kind functions. With this method, MPI automati- cally selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion
ticket230-B.	46	in heterogeneous environments. The second method (see Support for size-specific MPI Datatypes on page 667) gives the user complete control over communication by exposing

48machine representations.

Unofficial Draft for Comment Only

Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types — MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides representation conversion in heterogeneous environments. The mechanism described in this section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND parameter, where p is decimal digits of precision and r is an exponent range. Implicitly MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is defined for each value of (p, r) supported by the compiler, including pairs for which one value is unspecified. Attempting to access an element of the array with an index (p, r) not supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX datatypes. For integers, there is a similar implicit array related to selected_int_kind and indexed by the requested number of digits r. Note that the predefined datatypes contained in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but a new set.

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (*End of advice to implementors.*)

Advice to users. selected_real_kind() maps a large number of (p,r) pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same (p,r) value (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (*End of advice to users.*)

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI data type (handle)

int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: p, r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)

INTEGER P, R, NEWTYPE, IERROR

 $\mathbf{2}$

 24

 $45 \\ 46$

41 ticket-248T.

```
1
                 {static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r)(binding
            \mathbf{2}
                                 deprecated, see Section 15.2 }
            3
                      This function returns a predefined MPI datatype that matches a REAL variable of KIND
            4
                 selected_real_kind(p, r). In the model described above it returns a handle for the el-
            5
                 ement D(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r)
            6
                 (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. In communica-
            7
                 tion, an MPI datatype A returned by MPI_TYPE_CREATE_F90_REAL matches a datatype
            8
                 B if and only if B was returned by MPI_TYPE_CREATE_F90_REAL called with the same
            9
                 values for \mathbf{p} and \mathbf{r} or \mathbf{B} is a duplicate of such a datatype. Restrictions on using the returned
            10
                 datatype with the "external32" data representation are given on page 666.
            11
                      It is erroneous to supply values for p and r not supported by the compiler.
            12
            13
            14
                 MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)
            15
                   IN
                             р
                                                          precision, in decimal digits (integer)
            16
            17
                   IN
                             r
                                                          decimal exponent range (integer)
            18
                   OUT
                             newtype
                                                          the requested MPI datatype (handle)
            19
            20
                 int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
           21
ticket-248T.
            22
                 MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
            23
                      INTEGER, INTENT(IN) :: p, r
            24
                      TYPE(MPI_Datatype), INTENT(OUT) :: newtype
            25
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            26
                 MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
            27
                      INTEGER P, R, NEWTYPE, IERROR
            28
            29
                 {static MPI::Datatype MPI::Datatype::Create_f90_complex(int p,
            30
                                 int r) (binding deprecated, see Section 15.2) }
            31
                      This function returns a predefined MPI datatype that matches a
            32
                 COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from
            33
                 calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set
            34
                 to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to
            35
                 the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions
            36
                 on using the returned datatype with the "external32" data representation are given on page
            37
                 666.
            38
                      It is erroneous to supply values for p and r not supported by the compiler.
            39
            40
            41
                 MPI_TYPE_CREATE_F90_INTEGER(r, newtype)
            42
                   IN
                                                          decimal exponent range, i.e., number of decimal digits
                             r
            43
                                                          (integer)
            44
                   OUT
            45
                             newtype
                                                          the requested MPI datatype (handle)
            46
            47
                 int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
ticket-248T. 48
```

```
MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
                                                                                        1
                                                                                        2
    INTEGER, INTENT(IN) :: r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                                                                        4
                                           ierror
                                                                                        5
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
                                                                                        6
    INTEGER R, NEWTYPE, IERROR
                                                                                        8
{static MPI::Datatype MPI::Datatype::Create_f90_integer(int r) (binding
                                                                                        9
               deprecated, see Section 15.2 }
                                                                                        10
    This function returns a predefined MPI datatype that matches a INTEGER variable of
                                                                                        11
KIND selected_int_kind(r). Matching rules for datatypes created by this function are
                                                                                        12
analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
                                                                                        13
Restrictions on using the returned datatype with the "external 32" data representation are
                                                                                        14
given on page 666.
                                                                                        15
    It is erroneous to supply a value for r that is not supported by the compiler.
                                                                                        16
    Example:
                                                                                        17
                                                                                        18
   integer
                  longtype, quadtype
                                                                                        19
   integer, parameter :: long = selected_int_kind(15)
                                                                                        20
   integer(long) ii(10)
                                                                                       21
   real(selected_real_kind(30)) x(10)
                                                                                        22
   call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
                                                                                       23
   call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
                                                                                        24
   . . .
                                                                                        25
                                                                                        26
   call MPI_SEND(ii, 10, longtype, ...)
   call MPI_SEND(x, 10, quadtype, ...)
                                                                                       27
                                                                                        28
     Advice to users.
                        The datatypes returned by the above functions are predefined
                                                                                       29
```

Advice to users. The datatypes returned by the above functions are predefined datatypes. They cannot be freed; they do not need to be committed; they can be used with predefined reduction operations. There are two situations in which they behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

- 1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to retrieve the values of **p** and **r**.
- 2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI_TYPE_CREATE_F90_xxxx routines.

If a variable was declared specifying a non-default KIND value that was not obtained with selected_real_kind() or selected_int_kind(), the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section.

(End of advice to users.)

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_xxxx with the same combination of (xxxx,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To

Unofficial Draft for Comment Only

```
^{38}_{39} ticket250-V.
```

30

 31

32

33 34

35

36

37

40

41

42

43

44

45 46

47

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10 11

12

13 14

15

16

27

28

29

30

 31

32 33

34

35

ticket250-V.

prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_xxxx and using a hash[-] table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (xxxx,p,r). (*End of advice to implementors.*)

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2 on page 560) or user-defined (Section 13.5.3 on page 561) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.5.2 on page 560.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double" and "Double Extended" formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules.

```
<sup>26</sup> For MPI_TYPE_CREATE_F90_REAL:
```

```
if (p > 33) or (r > 4931) then external32 representation
is undefined
else if (p > 15) or (r > 307) then external32_size = 16
else if (p > 6) or (r > 37) then external32_size = 8
else external32_size = 4
```

For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL. For MPI_TYPE_CREATE_F90_INTEGER:

```
if
                (r > 38) then external32 representation is undefined
36
        else if (r > 18) then external32_size =
                                                   16
37
        else if (r > 9) then external32_size =
                                                   8
38
        else if (r > 4) then
                               external32_size =
                                                   4
39
        else if (r > 2) then
                               external32_size =
                                                   2
40
        else
                                external32_size =
                                                   1
41
```

⁴² If the external32 representation of a datatype is undefined, the result of using the datatype ⁴³ directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) ⁴⁴ in operations that require the external32 representation is undefined. These operations in-⁴⁶ clude MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions, ⁴⁷ when the "external32" data representation is used. The ranges for which the external32 ⁴⁸ representation is undefined are reserved for future standardization.

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI_<TYPE>n in C and Fortran and of the form MPI::<TYPE>n in C++ where <TYPE> is one of REAL, INTEGER and COMPLEX, and **n** is the length in bytes of the machine representation. This datatype locally matches all variables of type (**typeclass**, **n**). The list of names for such types includes:

MPI_REAL4		
MPI_REAL8		
MPI_REAL16		
MPI_COMPLEX8		
MPI_COMPLEX16		
MPI_COMPLEX32		
MPI_INTEGER1		
MPI_INTEGER2		
MPI_INTEGER4		
MPI_INTEGER8		
MPI_INTEGER16		

One datatype is required for each representation supported by the compiler. To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL*n, INTEGER*n, always create a variable whose representation is of size n. These datatypes may also be used for variables declared with KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV intrinsic module. Note that the MPI datatypes and the REAL*n, INTEGER*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined.

The following functions allow a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

MPI_SIZEOF(x, size)

OUTsizesize of machine representation of that type (integer)383940ticket-248T.MPI_Sizeof(x, size, ierror) BIND(C)41TYPE(*), DIMENSION() :: x42INTEGER, INTENT(OUT) :: size43INTEGER, OPTIONAL, INTENT(OUT) :: ierror44MPI_SIZEOF(X, SIZE, IERROR)45	IN	х	a Fortran variable of numeric intrinsic type (choice)	37
MPI_Sizeof(x, size, ierror) BIND(C)40 ticket-248T.TYPE(*), DIMENSION() :: x41INTEGER, INTENT(OUT) :: size42INTEGER, OPTIONAL, INTENT(OUT) :: ierror44	OUT	size	size of machine representation of that type (integer)	
TYPE(*), DIMENSION() :: x41INTEGER, INTENT(OUT) :: size42INTEGER, OPTIONAL, INTENT(OUT) :: ierror4344			_	
INTEGER, INTENT(OUT) :: size43INTEGER, OPTIONAL, INTENT(OUT) :: ierror44				41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44				42
44) ·· ierror	43
MPT_STZEOF(X_STZE, TERROR) 45				44
	MPI_SIZEO	F(X, SIZE, IERROR)		
<type> X 46</type>	51			-
INTEGER SIZE, IERROR 47	INTEG	ER SIZE, IERROR		

 $\mathbf{2}$

²⁸ ticket250-V.

	1 2			ze in bytes of the machine representation of the gi outine and has a Fortran binding only.	iven	
	3					
	4 5			function is similar to the C and $C++$ size of operator If given an array argument, it returns the size of the b		
	6		0 1 1	whole array. (End of advice to users.)	Jabe	
	7	Date	ionala This function is	not available in other languages because it would not	t ha	
	8 9		ful. (End of rationale.)	not available in other languages because it would not	t be	
	10					
	11					
ticket252-W		MPI_TYP	E_MATCH_SIZE(typecla	ass, size, <mark>data</mark> type)		
	13 14	IN	typeclass	generic type specifier (integer)		
	15	IN	size	size, in bytes, of representation (integer)		
	16 17	OUT	[ticket252-W.] <mark>data</mark> type	e datatype with correct type, size (handle)		
ticket252-W		int MPI_	Type_match_size(int	typeclass, int size, MPI_Datatype *datatype)		
ticket-248T	• 20 21 22 23	<pre>MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C) INTEGER, INTENT(IN) :: typeclass, size TYPE(MPI_Datatype), INTENT(OUT) :: datatype INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
ticket252-W ticket252-W	25		_MATCH_SIZE(TYPECLAS GER TYPECLASS, SIZE,	S, SIZE, <mark>DATA</mark> TYPE, IERROR) DATATYPE, IERROR		
	27 28	{static }		<pre>atatype::Match_size(int typeclass, ng deprecated, see Section 15.2) }</pre>		
	29 30 31 32 33 34 35 36 37 38 39 40	MPI_TYPE an MPI da This a duplicat size-specifi in order to suitable d MPI_SIZE by a call to	ECLASS_COMPLEX, correct atatype matching a local function returns a referen- te. This type cannot be a fic type that matches a F o compute the variable s latatype. In C and C+- EOF. In addition, for variable	ECLASS_REAL, MPI_TYPECLASS_INTEGER and esponding to the desired typeclass . The function retu l variable of type (typeclass , size). nce (handle) to one of the predefined named datatypes, freed. MPI_TYPE_MATCH_SIZE can be used to obta Fortran numeric intrinsic type by first calling MPI_SIZE size, and then calling MPI_TYPE_MATCH_SIZE to fir +, one can use the C function sizeof(), instead of iables of default kind the variable's size can be compu- TENT, if the typeclass is known. It is erroneous to spe- iler.	not in a EOF nd a 1ted	
	41 42			nience function. Without it, it can be tedious to find te to implementors below. (<i>End of rationale.</i>)	the	
	43 44 45	Adv	ice to implementors. T	his function could be implemented as a series of tests	-	
	46 47	int {	MPI_Type_match_size	(int typeclass, int size, MPI_Datatype *rtype	3)	
	48		<pre>witch(typeclass) {</pre>			

```
case MPI_TYPECLASS_REAL: switch(size) {
    case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
    case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
    default: error(...);
    }
    case MPI_TYPECLASS_INTEGER: switch(size) {
      case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
      case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
      default: error(...);
    }
    ... etc. ...
}
```

(End of advice to implementors.)

Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype $MPI_{<TYPE>n}$ can be received with this same datatype on another process. Most modern computers use 2's complement for integers and IEEE format for floating point. Thus, communication using these size-specific datatypes will not entail loss of precision or truncation errors.

Advice to users. Care is required when communicating in a heterogeneous environment. Consider the following code:

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
    ... initialize x ...
    call MPI_SEND(x, xtype, 100, 1, ...)
else if (myrank .eq. 1) then
    call MPI_RECV(x, xtype, 100, 0, ...)
endif
```

This may not work in a heterogeneous environment if the value of size is not the same on process 1 and process 0. There should be no problem in a homogeneous environment. To communicate in a heterogeneous environment, there are at least four options. The first is to declare variables of default type and use the MPI datatypes for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second is to use selected_real_kind or selected_int_kind and with the functions of the previous section. The third is to declare a variable that is known to be the same size on all architectures (e.g., selected_real_kind(12) on almost all compilers will result in an 8-byte representation). The fourth is to carefully check representation size before communication. This may require explicit conversion to a variable of size that can be communicated and handshaking between sender and receiver to agree on a size.

Unofficial Draft for Comment Only

 $\mathbf{2}$

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
4
                       real(selected_real_kind(5)) x(100)
            5
                       call MPI_SIZEOF(x, size, ierror)
            6
                       call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
            7
            8
                       if (myrank .eq. 0) then
            9
                           call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                                                            &
            10
                                                                                            &
                                                 MPI_MODE_CREATE+MPI_MODE_WRONLY,
            11
                                                 MPI_INFO_NULL, fh, ierror)
            12
                           call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
            13
                                                     MPI_INFO_NULL, ierror)
            14
                           call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
            15
                           call MPI_FILE_CLOSE(fh, ierror)
            16
                       endif
            17
            18
                       call MPI_BARRIER(MPI_COMM_WORLD, ierror)
            19
            20
                       if (myrank .eq. 1) then
            21
                           call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
                                                                                              X.
            22
                                           MPI_INFO_NULL, fh, ierror)
            23
                           call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
            24
                                                     MPI_INFO_NULL, ierror)
            25
                           call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
            26
                           call MPI_FILE_CLOSE(fh, ierror)
            27
                       endif
            28
            29
            30
                       If processes 0 and 1 are on different machines, this code may not work as expected if
            31
                       the size is different on the two machines. (End of advice to users.)
            32
            33
                          Problems With Fortran Bindings for MPI
                  16.2.10
            34
            35
                  This section discusses a number of problems that may arise when using MPI in a Fortran
            36
                  program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It
            37
                  does not add to the standard, but is intended to clarify the standard.
            38
                      As noted in the original MPI specification, the interface violates the Fortran standard
ticket
230-B. ^{39}
                  in several ways. While these may cause few problems for Fortran 77 programs, they become
            40
                  more significant for Fortran 90 programs, so that users must exercise care when using new
ticket230-B. ^{41}
                  Fortran 90 features. With Fortran 2008 and the new semantics defined in TR 29113, most
            42
                  violations are resolved, and this is hinted at in an addendum to each item. The violations
            43
                  were originally adopted and have been retained because they are important for the usability
ticket
235-G. ^{44}
                  of MPI. The rest of this section describes the potential problems in detail. It supersedes
            45
                  and replaces the discussion of Fortran bindings in the original MPI specification (for Fortran
            46
                  90, not Fortran 77).
ticket
230-B. ^{\rm 47}
                      The following MPI features are inconsistent with Fortran 90 and Fortran 77.
            48
```

Unofficial Draft for Comment Only

1

2

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi_f08 module together with a compiler that supports Fortran 2008 + TR 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. [Many]Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
- 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TR 29113.
- 5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_UNWEIGHTED, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 15 for more information.
- 6. The memory allocation routine MPI_ALLOC_MEM can't be usefully used in Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 - MPI-2.2. In Fortran 2003, TYPE(C_PTR) entities were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers.

Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used instead.

MPI-1 contained several routines that take address-sized information as input or return address-sized information as output. In C such arguments were of type MPI_Aint and in Fortran of type INTEGER. On machines where integers are smaller than addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking INTEGER arguments of KIND=MPI_ADDRESS_KIND. 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7 8

10

11 12 13

14

15

18

19

22

23

 24

25

26

27

28

ticket230-B.

ticket230-B. 9

ticket 247-S. 16

ticket 235-G. $^{\rm 17}$

ticket 235-G. 20

ticket 235-G. 21

ticket 235-G. 34

ticket 235-G. 35

ticket 235-G. 36

ticket235-G. 37

ticket235-G. 38

ticket235-G. 39

A number of new $\mathsf{MPI-2}$ functions also take $\mathtt{INTEGER}$ arguments of non-default <code>KIND</code>. See

Section 2.6 on page 17 and Section 4.1.1 on page 93 for more information.

Sections 16.2.11 through 16.2.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 16.2.7 on page 659.

16.2.11 Problems Due to Strong Typing

Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 16.2.6 on page 656). In C, the use of void* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TR 29113 (and later) together with the mpi_f08 module, the problem is avoided by declaring choice arguments with TYPE(*), DIMENSION(...), i.e., as assumed-type and assumed-rank dummy arguments.

[The]Using INCLUDE mpif.h, the following code fragment is technically [illegal]invalid and may generate a compile-time error.

integer i(5)
real x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)

ticket235-G.²⁹ In practice, it is rare for compilers to do more than issue a warning[, though there is ticket235-G.³⁰ Concern that Fortran 90 compilers are more likely to return errors]. Using the mpi_f08 or mpi module, the problem is usually resolved through the assumed-type and assumedrank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

> It is also technically [illegal]invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi_f08 or mpi module, the following code fragment[may generate] usually generates an error since the[buf argument] dims and periods arguments to [MPI_SEND is declared as an assumed-size array <type> buf(*)]MPI_CART_CREATE are declared as assumed size arrays INTEGER :: DIMS(*) and LOGICAL :: PERIODS(*).

```
ticket235-G. 40
                      [integer a
ticket235-G.<sup>41</sup>
                 call mpi_send(a, 1, MPI_INTEGER, ...) ]
            42
                   USE mpi_f08
                                      ! or USE mpi
            43
                    INTEGER size
            44
                   CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
            45
            46
                 Although this is a non-conforming MPI call, compiler warnings are not expected (but may
            47
                 occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit
```

ticket235-G. 48

interfaces.

ticket230-B.

	1
Advise to users. In the event that you mup into any of the problems related to targe	2
Advice to users. In the event that you run into one of the problems related to type checking, you may be able to work around it by using a compiler flag, by compiling	3
separately, or by using an MPI implementation with Extended Fortran Support as de-	4
scribed in Section 16.2.3. An alternative that will usually work with variables local to a	5
routine but not with arguments to a function or subroutine is to use the EQUIVALENCE	6
statement to create another variable with a type accepted by the compiler. (End of	7
advice to users.)	8
	9 10
]	11
	12
16.2.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets	¹² ticket236-H. ¹³ ticket230-B.
	15
Problems Due to Data Copying and Sequence Association	16
	17
] Arrays with subscript triplets describe Fortran subarrays with or without strides, e.g.,	$_{18}$ ticket236-H.
REAL a(100,100,100)	19
CALL MPI_Send(a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL,)	20
GALL MILBERG(a(11.17, 12.33.3, 1.100), 7+30+100, MILBER,)	21
The handling of subscript triplets depends on the value of the constant	22
MPI_SUBARRAYS_SUPPORTED:	23
	24
• If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:	25 26
Choice buffer arguments are declared as TYPE(*), DIMENSION(). For example,	27
consider the following code fragment:	28
	29
REAL s(100), r(100)	30
CALL MPI_Isend(s(1:100:5), 3, MPI_REAL,, rq, ierror)	31
CALL MPI_Wait(rq, status, ierror)	32
CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL,, rq, ierror)	33
CALL MPI_Wait(rq, status, ierror)	34
	35
In this case, the individual elements $s(1)$, $s(6)$, and $s(11)$ are sent between the start	36
of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy	37
s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code	38
will pass a descriptor to MPI_ISEND that allows MPI to operate directly on $s(1), s(6),$	39
$s(11),, s(96)$. The called MPI_ISEND routine will take only the first three of these	40
elements due to the type signature "3, MPI_REAL".	41
All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT,	42
MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice	43
buffers are copied to a contiguous scratch buffer in the MPI runtime environment.	44
All datatype descriptions (in the example above, "3, MPI_REAL") read and store	45
data from and to this virtual contiguous scratch buffer. Displacements in MPI de-	46
rived datatypes are relative to the beginning of this virtual contiguous scratch buffer.	47
	48

	674 CHAPTER 16. LANGUAGE BINDINGS
1 2 3 4 5 6	Upon completion of a nonblocking receive operation (e.g., when MPI_WAIT on a corresponding MPI_Request returns), it is as if the received data has been copied from the virtual contiguous scratch buffer back to the non-contiguous application buffer. In the example above, $r(1)$, $r(6)$, and $r(11)$ are guaranteed to be defined with the received data when MPI_WAIT returns.
6 7 8 9 10 11 12 13	Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION() arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and released this buffer not before the nonblocking commin- cation has completed (e.g., in an MPI wait routine). Efficient implementations may avoid such additional memory-to-memory data copying. (End of advice to implementors.)
14 15 16 17 18 19 20	<i>Rationale.</i> If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside of the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (<i>End of rationale.</i>)
21 22 23 24 25 26 27	• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.: Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.
ticket236-H. 28 ticket236-H. 29 30 31 32 ticket236-H. 33 34 35	In Fortran [90], [user]array data is not necessarily stored contiguously. For example, the array section $A(1:N:2)$ involves only the elements of A with indices 1, 3, 5, The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., $B(N)$) or is of assumed size (e.g., $B(*)$). If necessary, they do this by making a copy of the array into contiguous memory.[Both Fortran 77 and Fortran 90 are carefully worded to allow such copying to occur, but few Fortran 77 compilers do it.] ¹
ticket236-H. $\frac{36}{37}$ 38 39 40 41	Because MPI dummy buffer arguments are assumed-size arrays if MPI_SUBARRAYS_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:
42 43 44	real a(100) call MPI_IRECV(a(1:100:2), MPI_REAL, 50,)
45 46 47	Since the first dummy argument to MPI_IRECV is an assumed-size array (<type>buf(*)), the array section a(1:100:2) is copied to a temporary before being passed ¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless</type>

⁴⁷ ¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless ⁴⁸ the dummy argument has the CONTIGUOUS attribute.

to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array **a**. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for MPI_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a '[simple]simply contiguous' section such as A(1:N) of such an array. ([We define 'simple' more fully]'Simply contiguous' is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

[Our formal definition of a 'simple']According to the Fortran 2008 Standard, Section 6.5.4, a 'simply contiguous' array section is

```
name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )
```

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. The compiler can detect from analyzing the source code that the array is contiguous. Examples are

A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)

Because of Fortran's column-major ordering, where the first index varies fastest, a '[simple]simply contiguous' section of a contiguous array will also be contiguous.

Footnoteremoved:

To keep the definition of 'simply contiguous' simple, we have chosen to require all but one of the section subscripts to be without bounds. A colon without bounds makes it obvious both to the compiler and to the reader that the whole of the dimension is selected. It would have been possible to allow cases where the whole dimension is selected with one or two bounds, but this means for the reader that the array declaration or most recent allocation has to be consulted and for the compiler that a run-time check may be required.]

The same problem can occur with a scalar argument. [Some]A compiler[s, even for Fortran 77,] may make a copy of[some] scalar dummy arguments within a called procedure when passed as an actual argument to a choice buffer routine. That this can cause a problem is illustrated by the example

[ticket236-H.]real :: a
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a
subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end

44 45

46

47

	676 CHAPTER 16. LANGUAGE BINDINGS
1 2	If a is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter a itself.
3 4 5 6 7	Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., $A(1:n:2)$), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.
ticket236-H. $_9$ ticket236-H. $_{10}$	If there is a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this [should]must be employed.
ticket236-H. 11 ticket229.4. 12 ticket236-H. 13 ticket236-H. 14 15 16 17 ticket236-H. 18	If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, '[simple]simply contiguous' array sections of such arrays, or scalars, and if [there is no compiler option to inhibit this]no compiler option exists to inhibit such copying, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.
19 20 ticket236-H. 21 ticket236-H. 22 23 24	 16.2.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts Fortran arrays with vector subscripts describe subarrays containing a possibly irregular set of elements
25 26	REAL a(100) CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL,)
27 28 29 ticket230-B. 30	Arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.
31 32 ticket230-B. $^{32}_{33}$ 34	16.2.14 Special Constants [Special Constants
35 36 37 38 39 40] MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4 on page 15. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.
ticket250-V. 41 ticket250-V. 42 43 ticket182. 44 45	In Fortran, [the implementation of these special constants may require the use of lan- guage constructs that are outside the Fortran standard. Using]using special values for the constants (e.g., by defining them through parameter statements) is not possible because an implementation cannot distinguish these values from [legal]valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared
46 ticket250-V. 47 48	COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, [this address can be extracted by some mechanism outside the Fortran stan- dard (e.g., by Fortran extensions or by implementing the function in C)]the address of the

actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

16.2.15 Fortran Derived Types

```
Fortran 90 Derived Types
```

MPI[does not explicitly] supports passing Fortran[90] entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute. [Indeed, for MPI implementations that provide explicit interfaces through the mpi module a compiler will reject derived type actual arguments at compile time. Even when no explicit interfaces are given, users should be aware that Fortran 90 provides no guarantee of sequence association for derived types or arrays of derived types. For instance, an array of a derived type consisting of two elements may be implemented as an array of the first elements followed by an array of the second. Use of the SEQUENCE attribute may help here, somewhat.]

The following code fragment shows [one possible way to send a]some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type[ticket237-I.], BIND(C) :: mytype
                                                                                28
   integer [ticket229.2.]:: i
                                                                                29
   real [ticket229.2.]:: x
                                                                                30
   double precision [ticket229.2.]:: d
                                                                                31
   [ticket229.2.]logical :: 1
                                                                                32
                                                                                33
end type mytype
                                                                                34
type(mytype) [ticket250-V.]:: foo[ticket237-I.], fooarr(5)
                                                                                35
integer [ticket250-V.]:: blocklen(4), type(4)
                                                                                36
integer(KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: disp(4), base[ticket237-I.]37 lb, extent
                                                                                38
call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
                                                                                39
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
                                                                                40
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
                                                                                41
[ticket229.2.]call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
                                                                                42
                                                                                43
base = disp(1)
                                                                                44
disp(1) = disp(1) - base
                                                                                45
disp(2) = disp(2) - base
                                                                                46
disp(3) = disp(3) - base
                                                                                47
[ticket229.2.]disp(4) = disp(4) - base
                                                                                48
```

1

3

4

5

6

9 10

12

13 14

21

22

23

 24

25

26

27

² ticket242-N.

⁷ ticket230-B.
⁸ ticket237-I.

¹¹ ticket230-B.

¹⁵ ticket237-I.

¹⁶ ticket237-I.

¹⁷ ticket237-I.

¹⁸ ticket237-I.

¹⁹ ticket237-I.

²⁰ ticket237-I.

ticket237-I.

```
1
           \mathbf{2}
                     blocklen(1) = 1
           3
                     blocklen(2) = 1
           4
                     blocklen(3) = 1
           5
                     [ticket229.2.]blocklen(4) = 1
           6
           7
                     type(1) = MPI_INTEGER
            8
                     type(2) = MPI_REAL
           9
                     type(3) = MPI_DOUBLE_PRECISION
           10
                     [ticket229.2.]type(4) = MPI_LOGICAL
           11
           12
                     call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
           13
                     call MPI_TYPE_COMMIT(newtype, ierr)
           14
           15
                 [ticket237-I.] [! unpleasant to send foo% i instead of foo, but it works for scalar]
           16
                 [ticket237-I.][! entities of type mytype]
           17
                     call MPI_SEND(foo%i, 1, newtype, ...)
           18
                 [ticket237-I.]! or
           19
                                     call MPI_SEND(foo, 1, newtype, ...)
                 [ticket237-I.]
           20
                 [ticket237-I.]
                                     ! expects that base == address(foo%i) == address(foo)
           21
           22
                 [ticket237-I.]
                                     call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
           23
                 [ticket237-I.]
                                     call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
           ^{24}
                 [ticket237-I.]
                                     extent = disp(2) - disp(1)
           25
                 [ticket237-I.]
                                     1b = 0
           26
                                     call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
                 [ticket237-I.]
           27
                 [ticket237-I.]
                                     call MPI_TYPE_COMMIT(newarrtype, ierr)
           28
                 [ticket237-I.]
           29
                 [ticket237-I.]
                                     call MPI_SEND(fooarr, 5, newarrtype, ...)
ticket247-S. 30
                     Using the derived type variable foo instead of its first basic type element foo%i may
           31
                 be impossible if the MPI library implements choice buffer arguments through overloading
           32
                 instead of using TYPE(*), DIMENSION(...), or through a non-standardized extensions such
           33
ticket229.2. 34
                 as !$PRAGMA IGNORE_TKR; see Section 16.2.6 on page 656.
                     To use a derived type in an array requires a correct extent of the datatype handle to
           35
                 take care of the alignment rules applied by the compiler. These alignment rules may imply
           36
                 that there are gaps between the elements of a derived type, and also between the array
           37
ticket229.2. 38
                 elements. The extent of an iteroperable derived type (i.e., defined with BIND(C)) and a
                 SEQUENCE derived type with the same content may be different because C and Fortran may
           30
ticket229.2. 40
                 apply different alignment rules. [mytype is a SEQUENCE derived type. ]As recommended in
ticket229.2. 41
                 the advice to users in Section 4.1 on page 91, one should add an additional fifth structure
                 element with one numerical storage unit at the end of this structure to force in most cases
           42
                 that the array of structures is contiguous. Even with such an additional element, one should
           43
                 keep this resizing due to the special alignment rules that can be used by the compiler for
           44
                 structures, as also mentioned in this advice.
           45
                     Using the extended semantics defined in TR 29113, it is also possible to use entities
           46
                 or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer
           47
                 arguments; some additional constraints must be observed e.g., no ALLOCATABLE or POINTER
           48
```

type components may exist. In this case, the **base** address in the example must be changed to become the address of **foo** instead of **foo%i**, because the Fortran compiler may rearrange type components or add padding as it may fit for such types. Sending the structure **foo** should then also be performed by providing it (and not **foo%i**) as actual argument for MPI_Send.

16.2.16 Optimization Problems, an Overview

A Problem with Register Optimization

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. [This section discusses register optimization pitfalls.] These problems are independent of the Fortran support method; i.e., they occur with the mpi_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (Nonbl.).
- Use of one-sided routines (1-sided).
- Use of MPI parallel file I/O split collective operations (Split).
- Use of MPI_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 16.2.17 on page 680.
- Temporary data movement and temporary memory modifications; see Section 16.2.18 on page 687.
- Permanent data movement (e.g., through garbage collection); see Section 16.2.19 on page 688.

Table 16.4 shows in which usage areas the optimization problems may only occur. The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" to "VOLATILE" on pages 682-683,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 16.2.7 on page 659.

Unofficial Draft for Comment Only

⁵ ticket230-B.

⁶ ticket238-J.

¹⁷ ticket238-J. ¹⁸ ticket238-J.

```
^{19} ticket238-J.
```

20

21

22

23 24

25 26

27

28

29 30

 31

32 33

34

35

36 37

38

39 40

41

42

43

44 45

46 47

48

1

 $\mathbf{2}$

3

ticket238-J. ticket238-J.

	1	Optimization .			ay cause llowing u	-	
	3			Nonbl.	1-sided		Bottom
	4	Code movemen	nt	yes	yes	no	yes
	5	and register op		5	J -~		5.00
	6	Temporary dat		yes	yes	yes	no
	7	Permanent dat	a movement	yes	yes	yes	yes
	8						'
	9 10 11	Table 16.4: Occurrence	e of Fortran op	ptimizatio	on proble	ms in se	everal usage areas
	12 13	16.2.17 Problems with Co	de Movement	and Reg	gister Op	otimizat	ion
	14	Nonblocking operations					
ticket238-J	16 17 18 19 20	If a variable is local to a For- compiler will assume that it of argument of the call. In the to save and restore certain re- held a valid copy of such a v	cannot be moo most commo egisters. Thus	dified by a n linkage , the opti	a called s conventi mizer wi	ubroutin on, the ll assum	ne unless it is an actua subroutine is expected that a register which
	21 22	Example 16.11 Fortran 90	register optin	nization –	extreme		
	23	Source	compiled as		0	r compi	led as
	24 25	<pre>[ticket238-J.]REAL :: buf, call MPI_IRECV(buf,req)</pre>	-	ECV(buf,.	: buf, b .req) c	1	REAL :: buf, b [_IRECV(buf,req)
	26 27 28	call MPI_WAIT(req,) b1 = buf	call MPI_WAT	IT(req,			_WAIT(req,)
ticket238-J	29 30 31 32 33 34 . 35	Example 16.11 shows ex thread modifies buf between But the compiler cannot see returned, and may schedule has no reason to avoid using reorder the instructions as [i	any possibility the load of but a register to h	on of MP by that but if earlier the cold buf a	PI_IRECV f can be than type cross the	and th changed ed in th call to b	e finish of MPI_WAIT d after MPI_IRECV ha e source. The compile MPI_WAIT. It also may
	36 37	[ticket238-J.]					
	38	Example 16.12 Similar exa	mple with MI	PI_ISEND	1		
	39 40	Source	compiled as		W	-	ossible MPI-internal 1 sequence
	41 42 43	REAL :: buf, copy buf = val	REAL :: buf buf = val		F	EAL :: ouf = va	buf, copy l
	43 44 45	<pre>call MPI_ISEND(buf,req) copy = buf</pre>	<pre>call MPI_ISH copy= buf buf = val_ov</pre>		- 0	ddr = & opy= bu ouf = va	
	46 47	<pre>call MPI_WAIT(req,) buf = val_overwrite</pre>	call MPI_WA				ldr) ! within MPI_WAI
ticket228 I	48						

ticket 238-J. 48

Due to valid compiler code movement optimizations in Example 16.12, the content of buf may already be overwritten by the compiler when the content of buf is sent. The code movement is permitted because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization / code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication or parallel file I/O operation.

One-sided communication

An example with instruction reordering due to register optimization can be found in Section 11.7.4 on page 483.

MPI_BOTTOM and combining independent variables in datatypes

[Normally users are not afflicted with this. But the user should pay attention to this section if in his/her program]This section is only relevant if the MPI program uses a buffer argument to an MPI_SEND, MPI_RECV etc.,[uses a name] which hides the actual variables involved. MPI_BOTTOM with an MPI_Datatype containing absolute addresses is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one [mentioned]referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 16.13 shows what Fortran compilers are allowed to do.

Example 16.13 Fortran 90 register optimization.

This source	can be compiled as:	33
		34
call MPI_GET_ADDRESS(buf,bufaddr,	<pre>call MPI_GET_ADDRESS(buf,)</pre>	35
ierror)		36
<pre>call MPI_TYPE_CREATE_STRUCT(1,1,</pre>	call MPI_TYPE_CREATE_STRUCT()	37
bufaddr,		38
MPI_REAL,type,ierror)		
call MPI_TYPE_COMMIT(type,ierror)	<pre>call MPI_TYPE_COMMIT()</pre>	39
		40
val_old = buf	register = buf	41
	<pre>val_old = register</pre>	42
<pre>call MPI_RECV(MPI_BOTTOM,1,type,)</pre>	<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	43
<pre>val_new = buf</pre>	val_new = register	44
		"±"±

 45 ticket238-J.

⁴⁷ ticket238-J.

⁴⁸ ticket238-J.

[The]In Example 16.13, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access [of]to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10 11

12

14 15

16

17

19

20

21

22

23

24

25

26

28

29

30 31

32

¹³ ticket238-J.

18 ticket238-J.

ticket238-J.

ticket238-J.

ticket238-J.

	1	[ticket238-J.]					
	2 3	Example 16.14 Similar example with MPI_SEND					
	4	This source	can be compiled as:				
	5 6	! buf contains val_old	! buf contains val_old				
	7	buf = val_new	AND GEND()				
	8 9	<pre>call MPI_SEND(MPI_BOTTOM,1,type,) ! with buf as a displacement in type</pre>	<pre>call MPI_SEND() ! i.e. val_old is sent</pre>				
	9 10		1				
	11		<pre>! buf=val_new is moved to here ! and detected as dead code</pre>				
	12		and therefore removed				
	13 14		1				
	14	<pre>buf = val_overwrite</pre>	<pre>buf = val_overwrite</pre>				
	16						
	17	In Example 16.14, several successive assi	ignments to the same variable buf can be				
	18 19	combined in a way such that only the last assignment is executed. "Successive" means that					
	20	no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI_SEND statement is interfering because the load access to buf is hidden by the usage of MPI_BOTTOM.					
	21						
	22	to but is indden by the usage of with_both tow.					
ticket238-J.	23 24	Solutions					
ticket238-J.	25	The following sections show in detail how the	problems with code movement and register				
	26	optimization can be solved in a portable way.	Application writers can partially or fully				
	27	avoid these compiler optimization problems by					
	28 29	declarations with the send and receive buffers ations in which MPI_BOTTOM is used, or data					
	30	are used:	type nancies that combine several variables				
	31 32	• Use of the Fortran ASYNCHRONOUS attribu	ite.				
	33	• Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy					
	34	• Use of the helper fourne with 1 _3 + WC_KCG, of an equivalent user-written duminy routine.					
	35						
	36 37	• Declare the buffer as a Fortran module variable or within a Fortran common block.					
	38	• Use of the Fortran VOLATILE attribute.					
	39	Each of these methods solves the problems	of code movement and register optimization,				
	40	but may involve different degrees of performan	· · · · · · · · · · · · · · · · · · ·				
	41 42	application context. These methods may not he there must be guaranteed by a MDI 2.0 compliant					
	43	they must be guaranteed by a MPI-3.0 complian according to the requirements listed in Section					
	44	different impact on performance. MPI_F_SYN					
	45	and the $\texttt{ASYNCHRONOUS}$ attribute low through me	edium, and the VOLATILE attribute may have				
	46	the most negative impact on performance. Not					
	47 48	used for this purpose: the Fortran TARGET attrib in MPI applications.	oute does not solve code movement problems				
		TT					

The Fortran ASYNCHRONOUS attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping unit (or BLOCK) tells the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TR 29113 extension). Without the extensions specified in TR 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi_f08 module, with allowable exceptions listed in Section 16.2.6 on page 656), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if both the protection of the actual buffer argument through ASYNCHRONOUS according to the TR 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to .FALSE..

The ASYNCHRONOUS attribute has some restrictions. The TR 29113 defines (in the PDTR N1869):

"Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent. Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed. We are associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending com-

In Example 16.15 Case (a) on page 690, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read accesses to b occur after the communication completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to a pending communication affector while input communication (i.e., the two MPI_Irecv calls) 46 is pending. This is a contradiction to the rule that for input communication, a pending 47 communication affector shall not be referenced. The problem can be solved by using separate 48

Unofficial Draft for Comment Only

¹ ticket238-J. ² ticket238-J.

3

4

5

6

12

13

16

17

20

21 22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

38

39

40

41

42

43

44

7 8 9 10 11

 $^{14}_{15}$ ticket 229.1.

¹⁸ ₁₉ ticket238-J.

³⁷ ticket238-J.

684	CHAPTER 16. LANGUAGE BINDINGS
variables for the halos and the inner array, of subarrays which are passed through different of in Example 16.19 on page 692. If one does not overlap communication ar	
optimization problems can be solved through	the ASYNCHRONOUS attribute. wn in Example 16.13 and Example 16.14, can
	gument is defined as ASYNCHRONOUS to guaran-
[(Example 16.11 and its following paragr [To prevent instruction reordering or the two possibilities in portable Fortran code:]	raph were moved to an earlier position)] e allocation of a buffer in a register there are
Calling MPI_F_SYNC_REG	
The compiler may be prevented from movin MPI subroutine by surrounding the call by ca as an actual argument. The MPI library pro- purpose; see Section 16.2.8 on page 661.	alls to an external subroutine with the buffer
 The problems illustrated by the Examp MPI_F_SYNC_REG(buf) once immediat 	ples 16.11 and 16.12 can be solved by calling ely after MPI_WAIT.
Example 16.11	Example 16.12
<pre>can be solved with call MPI_IRECV(buf,req)</pre>	can be solved with buf = val
	<pre>call MPI_ISEND(buf,req) copy = buf</pre>
<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>
<pre>call MPI_F_SYNC_REG(buf) b1 = buf</pre>	<pre>call MPI_F_SYNC_REG(buf) buf = val_overwrite</pre>
	events moving the last line before the F_SYNC_REG(buf) are not needed because it ess copy=buf is moved below MPI_WAIT and
	ples 16.13 and 16.14 can be solved with two sements; one directly before MPI_RECV/ communication operation.

42		
43	Example 16.13	Example 16.14
44	can be solved with	can be solved with
45	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
46	call MPI_RECV(MPI_BOTTOM,)	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>
47	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
48		

1

 $\mathbf{2}$

3 4

 $\mathbf{5}$ 6

8

9

12

1314

16

17

18

22

36

3738

39 4041

ticket229.1. ⁷

ticket 238-J. $^{\scriptscriptstyle 10}$ ticket238-J. 11

ticket 238-J. 15

ticket238-J. 19

ticket 238-J. $_{\rm 20}$ ticket238-J. 21

The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store references to buf prior to MPI_RECV/MPI_SEND; the second call is needed to assure that the subsequent access to buf are not moved before MPI_RECV/SEND.

• In the example in Section 11.7.4 on page 483, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 16.11, i.e., a call to MPI_F_SYNC_REG(bbbb) is needed after the second MPI_WIN_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI_WIN_FENCE. In Process 2, both calls to MPI_WIN_FENCE together act as a communication call with MPI_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

Source of Process 1	Source of Process 2	:
bbbb = 777	buff = 999	:
	<pre>call MPI_F_SYNC_REG(buff)</pre>	:
call MPI_WIN_FENCE	call MPI_WIN_FENCE	:
call MPI_PUT(bbbb		:
into buff of process 2)		2
		2
call MPI_WIN_FENCE	call MPI_WIN_FENCE	2
call MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)	2
	ccc = buff	2
		(

• The temporary memory modification problem, i.e., Example 16.16 on page 691, can **not** be solved with this method.

A user defined routine instead of MPI_F_SYNC_REG

Instead of MPI_F_SYNC_REG, one can also use a user defined external subroutine, which is separately compiled: [with the separately compiled]

subroutine DD(buf)
integer buf
end

Π

Note that if the intent is declared in an explicit interface for the external subroutine, it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, [the above]a call [of]to MPI_RECV with MPI_BOTTOM as buffer might be replaced by

call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)

 $\mathbf{2}$

1 2 3	Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer
4 5 6	MPI_F_SYNC_REG or one of the other posibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented according to the rules specified in Section 16.2.7
ticket238-J. 7 8	on page 659. [(assuming that buf has type INTEGER). The compiler may be similarly prevented from
9 10 11	moving a reference to a variable across a call to an MPI subroutine. In the case of a nonblocking call, as in the above call of MPI_WAIT, no reference to the buffer is permitted until it has been verified that the transfer has been completed.
12 13	Therefore, in this case, the extra call ahead of the MPI call is not necessary, i.e., the call of MPI_WAIT in the example might be replaced by
¹⁴ ticket238-J. ¹⁵ ticket238-J. ¹⁶	/tt call MPI_WAIT(req,) call DD(buf)]
17 18	Module variables and COMMON blocks
ticket238-J. $_{20}^{19}$	An alternative to the already mentioned methods is to put the buffer or variable into a
21 22	module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine.
ticket238-J. ₂₃ ticket238-J. ₂₄ ticket238-J. ₂₅	The compiler will then have to assume that the MPI procedure [(MPI_RECV in the above example)]may alter the buffer or variable, provided that the compiler cannot [analyze]infer that the MPI procedure does not reference the module or common block.
26 27 28	• This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
29 30 31 32	• Unfortunately, this method does not solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.
ticket238-J. 33 ticket238-J. 34	0
35 36	The (poorly performing) Fortran VOLATILE attribute
$ticket 238$ -J. 37 $ticket 238$ -J. $^{38}_{39}$ $_{40}$	The VOLATILE attribute[, available in later versions of Fortran,] gives the buffer or variable the properties needed, but it may inhibit optimization of any code containing references or definitions of the buffer or variable.
ticket238-J. 41	The Fortran TARGET attribute
ticket238-J. 42 43 44	The TARGET attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program appricate the TARGET attribute for an actual buffer argument used
45 46 47	the application program specifies the TARGET attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.
48	

Rationale. The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TR 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

16.2.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 16.15, Case (b) on page 690. Example 16.16 on page 691 shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 16.17, buf(100,100) from Example 16.16 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 16.18 shows a second possible optimization. The whole array is temporarily moved to local_buf. When storing local_buf back to the original location buf, then this includes also an overwriting of the receive buffer part buf(1,1:100), i.e., this storing back may overwrite the asynchronously received data.

Note, that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11 on page 421.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations with between the ..._BEGIN and ..._END calls; see Section 13.4.5 on page 548.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 683 ³² in Section 16.2.17 on page 680, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the numerical read ³⁴ accesses are separated into different variables, as shown in Example 16.19 on page 692 and ³⁵ in Example 16.20 on page 693. ³⁶

Note also that the methods

- calling MPI_F_SYNC_REG (or such a user-defined routine),
- using module variables and COMMON blocks, and
- the TARGET attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the numerical code shown in Example 16.16 and 16.17.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

19

20

21

22

23 24

> 25 26

27

28 29

30

31

37 38

39 40

41 42

43

44

45

46

⁵ ⁶ ticket238-J. ⁷ ticket238-J. ⁸ ⁹ ¹⁰ ¹⁰ ticket238-J. ¹² ¹³ 1 any storage unit (word) of buf must be directly done in the main memory exactly in the $\mathbf{2}$ sequence defined by the application program. The VOLATILE attribute prevents all register 3 and cache optimizations. Therefore, VOLATILE may cause a huge performance degradation. 4 Instead of solving the problem, it is needed to **prevent** the problem. When overlapping $\mathbf{5}$ communication and computation, the nonblocking communication (or nonblocking or split 6 collective IO) and the computation should be executed on different sets of variables. 7In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI 8 9 operations.

Rationale. This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature is available in a later version of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (End of rationale.)

In Example 16.19 on page 692 (which is a solution for the problem shown in Example 16.15 on page 690) and in Example 16.20 on page 693 (which is a solution for the problem shown in Example 16.18 on page 691), the array is split into inner and halo part and both disjunct parts are passed to a subroutine **separated_sections**. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the calculation on the elements where inner+halo is needed. Note that the halo and the inner area are strided arrays. Those can be used in non-blocking communication only with a TR 29113 based MPI library.

29

30

 31

32

33 34

35

36

37

38 39

40

42 43

44

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 24

25

16.2.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. Automatic garbage collection implementation is one use case. Such permanent data movement is in conflict with MPI in several areas:

• MPI datatype handles with absolute addresses in combination with MPI_BOTTOM.

• Nonblocking MPI operations (communication, one-sided, I/O) if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 16.2.7 on page 659.

ticket238-J.⁴¹

16.2.20 Comparison with C

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the & operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not

ticket238-J.

occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while the nonblocking operations are pending.

```
1
2
3
4
5
6
7
     [ticket238-J.]
8
9
     Example 16.15 Protecting nonblocking communication with the ASYNCHRONOUS attribute.
10
11
     USE mpi_f08
12
13
     REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
     REAL :: bnew(0:101)
                                      ! elements 1 and 100 are newly computed
14
     TYPE(MPI_Request) :: req(4)
15
     INTEGER :: left, right, i
16
     CALL MPI_Cart_shift(...,left,right,...)
17
     CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
18
     CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
19
     CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
20
     CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)
21
22
     #ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
23
     ! Case (a)
24
       CALL MPI_Waitall(4,req,...)
25
       DO i=1,100 ! compute all new local data
26
         bnew(i) = function(b(i-1), b(i), b(i+1))
27
       END DO
28
     #endif
29
30
     #ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
^{31}
     ! Case (b)
32
       DO i=2,99 ! compute only elements for which halo data is not needed
33
         bnew(i) = function(b(i-1), b(i), b(i+1))
34
       END DO
35
       CALL MPI_Waitall(4,req,...)
36
       i=1 ! compute leftmost element
37
         bnew(i) = function(b(i-1), b(i), b(i+1))
38
       i=100 ! compute rightmost element
39
         bnew(i) = function(b(i-1), b(i), b(i+1))
40
     #endif
41
42
43
44
45
46
47
48
```

[ticket238-J.]	1 2
Example 16.16 Overlapping Communication and Computation.	3 4
NGE wai f00	4 5
USE mpi_f08 REAL :: buf(100,100)	6
CALL MPI_Irecv(buf(1,1:100),req,)	7
DO j=1,100	8
DO i=2,100	9
buf(i,j)=	10
END DO	11
END DO	12
CALL MPI_Wait(req,)	13
	14
	15 16
[ticket238-J.]	17
	18
Example 16.17 The compiler may substitute the nested loops through loop fusion.	19
REAL :: buf(100,100), buf_1dim(10000)	20
EQUIVALENCE (buf(1,1), buf_1dim(1))	21
CALL MPI_Irecv(buf(1,1:100),req,)	22
tmp(1:100) = buf(1,1:100)	23
DO j=1,10000	24
<pre>buf_1dim(h)=</pre>	25
END DO	26 27
buf(1,1:100) = tmp(1:100)	28
CALL MPI_Wait(req,)	29
	30
	31
[ticket238-J.]	32
	33
Example 16.18 Another optimization is based on the usage of a separate memory storage	34
area, e.g., in a GPU.	35
REAL :: buf(100,100), local_buf(100,100)	36
CALL MPI_Irecv(buf(1,1:100),req,)	37
local_buf = buf	38 39
DD j=1,100	40
DO i=2,100	41
<pre>local_buf(i,j)=</pre>	42
END DO	43
END DO	44
<pre>buf = local_buf ! may overwrite asynchronously received</pre>	45
CALL MPI_Wait(req)	46
	47
	48

```
1
2
3
4
5
6
     [ticket238-J.]
7
8
     Example 16.19 Using separated variables for overlapping communication and computation
9
     to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.
10
     USE mpi_f08
11
     REAL :: b(0:101)
                           ! elements 0 and 101 are halo cells
12
     REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
13
     INTEGER :: i
14
     CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
15
     i=1 ! compute leftmost element
16
       bnew(i) = function(b(i-1), b(i), b(i+1))
17
     i=100 ! compute rightmost element
18
       bnew(i) = function(b(i-1), b(i), b(i+1))
19
     END
20
21
     SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
22
     USE mpi_f08
23
     REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
24
     REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
25
     TYPE(MPI_Request) :: req(4)
26
     INTEGER :: left, right, i
27
     CALL MPI_Cart_shift(...,left,right,...)
28
     CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
29
     CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
30
     ! b_lefthalo and b_righthalo is written asynchronously.
^{31}
     ! There is no other concurrent access to b_lefthalo and b_righthalo.
32
     CALL MPI_Isend(b_inner( 1), ..., left, ..., req(3), ...)
33
     CALL MPI_Isend(b_inner(100),
                                    ..., right, ..., req(4), ...)
34
35
     DO i=2,99 ! compute only elements for which halo data is not needed
36
       bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
37
       ! b_inner is read and send at the same time.
38
       ! This is allowed based on the rules for ASYNCHRONOUS.
39
     END DO
40
     CALL MPI_Waitall(4,req,...)
41
     END SUBROUTINE
42
43
44
45
46
47
48
```

```
11
                                                                                       12
[ticket238-J.]
                                                                                       13
                                                                                       14
Example 16.20 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
                                                                                       15
                                                                                       16
USE mpi_f08
REAL :: buf(100,100)
                                                                                       18
CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
                                                                                       19
END
                                                                                       20
                                                                                       21
SUBROUTINE separated_sections(buf_halo, buf_inner)
                                                                                       22
REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
                                                                                       23
REAL :: buf_inner(2:100,1:100)
                                                                                       24
REAL :: local_buf(2:100,100)
                                                                                       25
                                                                                       26
CALL MPI_Irecv(buf_halo(1,1:100),...req,...)
                                                                                       27
local_buf = buf_inner
                                                                                       28
DO j=1,100
                                                                                       29
  DO i=2,100
                                                                                       30
    local_buf(i,j)=....
                                                                                       ^{31}
  END DO
                                                                                       32
END DO
                                                                                       33
buf_inner = local_buf ! buf_halo is not touched!!!
                                                                                       34
                                                                                       35
CALL MPI_Wait(req,...)
                                                                                       36
                                                                                       37
                                                                                       38
                                                                                       39
                                                                                       ^{41}
                                                                                       42
                                                                                       43
                                                                                       44
                                                                                       45
```

16.3 Language Interoperability

16.3.1 Introduction

It is not uncommon for library developers to use one language to develop an applications library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C, C++, and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

14 15

21

22

23

 24

25 26 27

1 2

3 4

5

6

7

8

9

10

11

12

13

Initialization We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.
 - **Interlanguage communication** We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

16.3.2 Assumptions

28We assume that conventions exist for programs written in one language to call routines 29 written in another language. These conventions specify how to link routines in different 30 languages into one program, how to call functions in a different language, how to pass ar- 31 guments between languages, and the correspondence between basic data types in different 32 languages. In general, these conventions will be implementation dependent. Furthermore, 33 not every basic datatype may have a matching type in other languages. For example, 34 C/C++ character strings may not be compatible with Fortran CHARACTER variables. How-35 ever, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array 36 of INTEGERS, can be passed to a C or C++ program. We also assume that Fortran, C, and 37 C++ have address-sized integers. This does not mean that the default-size integers are the 38 same size as default-sized pointers, but only that there is some way to hold (and pass) a 39 C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI_OFFSET_KIND) 40 can be passed from Fortran to C as MPI_Offset. 41

41 42

43

16.3.3 Initialization

⁴⁴ A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution
 ⁴⁵ in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of
 the C/C++ version of MPI_INIT in order to propagate values for argc and argv to

Unofficial Draft for Comment Only

all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI_INITIALIZED returns the same answer in all languages. The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

16.3.4 Transfer of Handles

Handles are passed between Fortran and C or C++ by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C or C++ handles in Fortran. Handles are passed between C and C++ using overloaded C++ operators called from C++ code. There is no direct access to C++ objects from C.

The type definition MPI_Fint is provided in C/C++ for an integer of the size that matches a Fortran INTEGER; usually, MPI_Fint will be equivalent to int. With the Fortran mpi module or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in the following conversion functions. With the Fortran mpi_f08 module, a Fortran handle is a BIND(C) derived type that contains an INTEGER field named MPI_VAL. This INTEGER value can be used in the following conversion functions.

The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.5 on page 23.

MPI_Comm MPI_Comm_f2c(MPI_Fint comm)

If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a valid C handle to that same communicator; if comm = MPI_COMM_NULL (Fortran value), then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then MPI_Comm_f2c returns an invalid C handle.

MPI_Fint MPI_Comm_c2f(MPI_Comm comm)

The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle to the same communicator; it maps a null handle into a null handle and an invalid handle into an invalid handle.

Similar functions are provided for the other types of opaque objects.

Unofficial Draft for Comment Only

 24

²⁸ ticket231-C.

```
1
              MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
         \mathbf{2}
              MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
         3
         4
              MPI_Group MPI_Group_f2c(MPI_Fint group)
         5
              MPI_Fint MPI_Group_c2f(MPI_Group group)
         6
         7
              MPI_Request MPI_Request_f2c(MPI_Fint request)
         8
              MPI_Fint MPI_Request_c2f(MPI_Request request)
         9
         10
              MPI_File MPI_File_f2c(MPI_Fint file)
         11
              MPI_Fint MPI_File_c2f(MPI_File file)
         12
         13
              MPI_Win MPI_Win_f2c(MPI_Fint win)
         14
              MPI_Fint MPI_Win_c2f(MPI_Win win)
         15
         16
              MPI_Op MPI_Op_f2c(MPI_Fint op)
         17
              MPI_Fint MPI_Op_c2f(MPI_Op op)
         18
         19
              MPI_Info MPI_Info_f2c(MPI_Fint info)
         20
        21
              MPI_Fint MPI_Info_c2f(MPI_Info info)
         22
              MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
         23
         ^{24}
              MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
ticket274. 25
              MPI_Message MPI_Message_f2c(MPI_Fint message)
         26
        27
              MPI_Fint MPI_Message_c2f(MPI_Message message)
         28
         29
              Example 16.21 The example below illustrates how the Fortran MPI function
         30
              MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function
         ^{31}
              MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C
         32
              interface is assumed where a Fortran function is all upper case when referred to from C and
         33
              arguments are passed by addresses.
         34
        35
              ! FORTRAN PROCEDURE
        36
              SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
        37
              INTEGER [ticket250-V.]:: DATATYPE, IERR
        38
              CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
         39
              RETURN
         40
              END
         41
        42
              /* C wrapper */
         43
         44
              void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)
         45
              {
         46
                 MPI_Datatype datatype;
         47
         48
                 datatype = MPI_Type_f2c( *f_handle);
```

Unofficial Draft for Comment Only

}

```
*ierr = (MPI_Fint)MPI_Type_commit( &datatype);
*f_handle = MPI_Type_c2f(datatype);
return;
```

The same approach can be used for all other MPI functions. The call to MPI_xxx_f2c (resp. MPI_xxx_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

Rationale. The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

C and C++ The C++ language interface provides the functions listed below for mixedlanguage interoperability. The token $\langle CLASS \rangle$ is used below to indicate any valid MPI opaque handle name (e.g., Group), except where noted. For the case where the C++ class corresponding to $\langle CLASS \rangle$ has derived classes, functions are also provided for converting between the derived classes and the C MPI_ $\langle CLASS \rangle$.

The following function allows assignment from a C MPI handle to a C++ MPI handle.

```
MPI:::<CLASS>& MPI:::<CLASS>::operator=(const MPI_<CLASS>& data)
```

The constructor below creates a C++MPI object from a C MPI handle. This allows the automatic promotion of a C MPI handle to a C++MPI handle.

MPI::<CLASS>::<CLASS>(const MPI_<CLASS>& data)

Example 16.22 In order for a C program to use a C++ library, the C++ library must export a C interface that provides appropriate conversions before invoking the underlying C++ library call. This example shows a C interface function that invokes a C++ library call with a C communicator; the communicator is automatically promoted to a C++ handle when the underlying C++ function is invoked.

```
// C++ library function prototype
                                                                                       40
void cpp_lib_call(MPI::Intracomm cpp_comm);
                                                                                       41
                                                                                       42
// Exported C function prototype
                                                                                       43
extern "C" {
                                                                                       44
   void c_interface(MPI_Comm c_comm);
                                                                                       45
}
                                                                                       46
                                                                                       47
void c_interface(MPI_Comm c_comm)
                                                                                       48
```

 24

 31

```
1
                  {
            \mathbf{2}
                     // the MPI_Comm (c_comm) is automatically promoted to MPI::Intracomm
            3
                     cpp_lib_call(c_comm);
            4
                  }
            5
                      The following function allows conversion from C++ objects to C MPI handles. In this
            6
                  case, the casting operator is overloaded to provide the functionality.
            7
            8
                 MPI::<CLASS>::operator MPI_<CLASS>() const
            9
            10
                  Example 16.23 A C library routine is called from a C++ program. The C library routine
            11
                  is prototyped to take an MPI_Comm as an argument.
            12
            13
                  // C function prototype
            14
                  extern "C" {
            15
                     void c_lib_call(MPI_Comm c_comm);
            16
                  }
            17
            18
                  void cpp_function()
            19
                  {
            20
                     // Create a C++ communicator, and initialize it with a dup of
            21
                           MPI::COMM_WORLD
                     //
            22
                     MPI::Intracomm cpp_comm(MPI::COMM_WORLD.Dup());
            23
                     c_lib_call(cpp_comm);
            24
                  }
            25
            26
                                    Providing conversion from C to C++ via constructors and from C++
                       Rationale.
            27
                       to C via casting allows the compiler to make automatic conversions. Calling C from
                       C++ becomes trivial, as does the provision of a C or Fortran interface to a C++
            28
            29
                       library. (End of rationale.)
            30
                       Advice to users. Note that the casting and promotion operators return new handles
            31
                       by value. Using these new handles as INOUT parameters will affect the internal MPI
            32
                       object, but will not affect the original handle from which it was cast. (End of advice
            33
                       to users.)
            34
            35
                      It is important to note that all C++ objects with corresponding C handles can be used
            36
                  interchangeably by an application. For example, an application can cache an attribute on
            37
                  MPI_COMM_WORLD and later retrieve it from MPI:::COMM_WORLD.
            38
            39
                  16.3.5 Status
            40
ticket243-O.<sup>41</sup>
                  The following two procedures are provided in C to convert from a Fortran (with the mpi
            42
                  module or mpif.h) status (which is an array of integers) to a C status (which is a structure),
            43
                  and vice versa. The conversion occurs on all the information in status, including that which
            44
                  is hidden. That is, no status information is lost in the conversion.
            45
  ticket140. 46
                  int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
            47
                      If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE
            48
                  or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with
```

the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE[, respectively.] defined in the mpi module or mpif.h. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

To do the conversion in the other direction, we have the following: int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)

This call converts a C status into a Fortran status, and has a behavior similar to MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE.

Advice to users. There [is not a]exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Fig. 16.1 on page 700. (*End of advice to users.*)

Rationale. The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, then the type of its result would have to be MPI_Status**, which was considered an inferior solution. (*End of rationale.*)

Using the mpi_f08 Fortran module, a status is declared as TYPE(MPI_Status). The C type MPI_F08_status can be used to pass a Fortran TYPE(MPI_Status) argument into a C routine. Figure 16.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.

This C routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a C MPI_Status.

This C routine converts a C MPI_Status into a Fortran mpi_f08 TYPE(MPI_Status). Two global variables of type MPI_F08_status*, MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in the mpi_f08 module. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

```
1
                                                         MPI Status
                              C types and functions
             2
             3
             4
             5
             6
             7
             8
             9
             10
                                                      MPI_Status_f2f08()
             11
                                MPI_F08_status
                                                                               MPI Fint array
            12
                                                      MPI_Status_f082f()
            13
                                 Equivalent types
                                                                               Equivalent types
            14
                               (identical memory layout)
                                                                            (identical memory layout)
             15
             16
             17
                                                      MPI_Status_f2f08()
                                                                             INTEGER array
                                TYPE(MPI_Status)
                                                                            of size MPI_STATUS_SIZE
             18
                                                      MPI_Status_f082f()
             19
                              Fortran types and subroutines
            20
            21
                                             Figure 16.1: Status conversion routines
            22
            23
            ^{24}
                       Conversion between the two Fortran versions of a status can be done with:
            25
            26
                   MPI_STATUS_F2F08(f_status, f08_status)
            27
            28
                     IN
                               f_status
                                                              status object declared as array
            29
                     OUT
                               f08_status
                                                              status object declared as named type
            30
            ^{31}
            32
                   int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
ticket-248T.
            33
                   MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C)
            34
                        INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
            35
                       TYPE(MPI_Status), INTENT(OUT) :: f08_status
            36
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
            37
                   MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
            38
            39
                        INTEGER :: F_STATUS(MPI_STATUS_SIZE)
                       TYPE(MPI_Status) :: F08_STATUS
            40
                       INTEGER IERROR
            41
            42
                       This routine converts a Fortran INTEGER, DIMENSION (MPI_STATUS_SIZE) status array
            43
                   into a Fortran mpi_f08 TYPE(MPI_Status).
            44
            45
            46
             47
             48
```

MPI_ST	ATUS_F082F(f08_sta	atus, f_status)	1
IN	f08_status	status object declared as named type	2
- · ·	_		3
OUT	f_status	status object declared as array	4
			5
int MPI	_Status_f082f(MPI	I_F08_status *f08_status, MPI_Fint *f_status)	6
107 G.			$_{7}$ ticket-248T.
		atus, f_status, ierror) BIND(C)	8
		NTENT(IN) :: f08_status	9
) :: f_status(MPI_STATUS_SIZE)	10
INT	EGER, OPTIONAL, I	INTENT(OUT) :: ierror	11
MPI_STA	TUS_F082F(F08_STA	ATUS, F_STATUS, IERROR)	12
TYF	PE(MPI_Status) ::	F08_STATUS	13
INT	EGER :: F_STATUS	S(MPI_STATUS_SIZE)	14
INT	EGER IERROR		15
			16
		Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER.	17
DIMENSI	ON(MPI_STATUS_SIZ	ZE) status array.	18

16.3.6 MPI Opaque Objects

Unless said otherwise, opaque objects are "the same" in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail, issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see 16.3.9, page 708).

Example 16.24

```
! FORTRAN CODE
REAL [ticket250-V.]:: R(5)
INTEGER [ticket250-V.]:: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: AODISP(1)
! create an absolute datatype for array R
```

 24

 31

```
1
     AOBLEN(1) = 5
\mathbf{2}
     CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
3
     AOTYPE(1) = MPI_REAL
4
     CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
5
     CALL C_ROUTINE(TYPE)
6
     /* C code */
7
8
9
     void C_ROUTINE(MPI_Fint *ftype)
10
     {
11
         int count = 5;
         int lens[2] = \{1, 1\};
12
        MPI_Aint displs[2];
13
14
        MPI_Datatype types[2], newtype;
15
         /* create an absolute datatype for buffer that consists
16
                                                                           */
         /* of count, followed by R(5)
17
                                                                           */
18
19
        MPI_Get_address(&count, &displs[0]);
        displs[1] = 0;
20
        types[0] = MPI_INT;
21
        types[1] = MPI_Type_f2c(*ftype);
22
23
        MPI_Type_create_struct(2, lens, displs, types, &newtype);
24
        MPI_Type_commit(&newtype);
25
26
        MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
         /* the message sent contains an int count of 5, followed
27
                                                                           */
         /* by the 5 REAL entries of the Fortran array R.
                                                                           */
28
     }
29
30
           Advice to implementors. The following implementation can be used: MPI addresses,
31
           as returned by MPI_GET_ADDRESS, will have the same value in all languages. One
32
           obvious choice is that MPI addresses be identical to regular addresses. The address
33
           is stored in the datatype, when datatypes with absolute addresses are constructed.
34
           When a send or receive operation is performed, then addresses stored in a datatype
35
           are interpreted as displacements that are all augmented by a base address. This base
36
           address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM
37
           is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly
38
           as a call with a regular buffer argument: in both cases the base address is buf. On the
39
           other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly
40
           different. A test is performed to check whether buf = MPI_BOTTOM. If true, then
41
           the base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does
42
           not have the same value in Fortran and C/C++, then an additional test for buf =
43
           MPI_BOTTOM is needed in at least one of the languages.
44
45
           It may be desirable to use a value other than zero for MPI_BOTTOM even in C/C++,
46
           so as to distinguish it from a NULL pointer. If MPI_BOTTOM = c then one can still
47
           avoid the test buf = MPI_BOTTOM, by using the displacement from MPI_BOTTOM,
48
```

i.e., the regular address - c, as the MPI address returned by MPI_GET_ADDRESS and stored in absolute datatypes. (End of advice to implementors.)

Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are associated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language and language support method), and is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI_COMM_NULL_COPY_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI_COMM_CREATE_KEYVAL); see also the advice to users on page 282. (End of advice to users.)

Error Handlers

Advice to implementors. Error handlers, have, in C and C++, a "stdargs" argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (End of advice to implementors.)

Reduce Operations

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C, C++, and Fortran datatypes. (End of advice to users.)

Addresses

Some of the datatype accessors and constructors have arguments of type MPI_Aint (in C) or MPI::Aint in C++, to hold addresses. The corresponding arguments, in Fortran, have type INTEGER. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran INTEGERs have 32 bits.

This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of ≥ 4 GB. What should be the value returned in Fortran by MPI_ADDRESS, for a variable with an address above 2^{32} ? The design described here addresses this issue, while maintaining compatibility with current Fortran codes.

Unofficial Draft for Comment Only

1

2

3 4

5

6

 $\overline{7}$

8

9

10

11

17

20

21

22

23

 24

25

262728

29

30

31

32 33

34 35

36

37

38 39

40 41

42

4344

4546

4748

1213 14 $_{15}$ ticket 229.1. 16 ₁₈ ticket230-B. ticket229.1. 19

¹ The constant MPI_ADDRESS_KIND is defined so that, in Fortran 90, ² INTEGER(KIND=MPI_ADDRESS_KIND)) is an address sized integer type (typically, but not ³ necessarily, the size of an INTEGER(KIND=MPI_ADDRESS_KIND) is 4 on 32 bit address ma-⁴ chines and 8 on 64 bit address machines). Similarly, the constant MPI_INTEGER_KIND is ⁵ defined so that INTEGER(KIND=MPI_INTEGER_KIND) is a default size INTEGER.

There are seven functions that have address arguments: MPI_TYPE_HVECTOR,
 MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT
 MPI_TYPE_LB and MPI_TYPE_UB.

⁹ Four new functions are provided to supplement the first four functions in this list. ¹⁰ These functions are described in Section 4.1.1 on page 93. The remaining three functions ¹¹ are supplemented by the new function MPI_TYPE_GET_EXTENT, described in that same ¹² section. The new functions have the same functionality as the old functions in C/C++, ¹³ or on Fortran systems where default INTEGERs are address sized. In Fortran, they accept ¹⁴ arguments of type INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type

¹⁵ MPI_Aint and MPI::Aint are used in C and C++. On Fortran 77 systems that do not support ¹⁶ the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs ¹⁷ are 32 bits, these arguments will be of an appropriate integer type. The old functions will ¹⁸ continue to be provided, for backward compatibility. However, users are encouraged to ¹⁹ switch to the new functions, in Fortran, so as to avoid problems on systems with an address ²⁰ range > 2^{32} , and to provide compatibility across languages.

²² 16.3.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.)

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

33

23

- 34 35
- Advice to implementors. This requires that attributes be tagged either as "C," "C++" or "Fortran," and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)
- 36 37 38

39

40

41

The attribute manipulation functions described in Section 6.7 on page 276 define attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C/C++ pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C/C++ callee, or vice-versa.

⁴² MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs ⁴³ are smaller, then the Fortran function MPI_ATTR_GET will return the least significant part ⁴⁴ of the attribute word; the Fortran function MPI_ATTR_PUT will set the least significant ⁴⁵ part of the attribute word, which will be sign extended to the entire word. (These two ⁴⁷ functions may be invoked explicitly by user code, or implicitly, by attribute copying callback ⁴⁸ functions.) As for addresses, new functions are provided that manipulate Fortran address sized 1 attributes, and have the same functionality as the old functions in C/C++. These functions 2 are described in Section 6.7, page 276. Users are encouraged to use these new functions. 3

4 MPI supports two types of attributes: address-valued (pointer) attributes, and integer valued attributes. C and C++ attribute functions put and get address valued attributes. 5Fortran attribute functions put and get integer valued attributes. When an integer valued 6 $\overline{7}$ attribute is accessed from C or C++, then MPI_xxx_get_attr will return the address of (a pointer to) the integer valued attribute, which is a pointer to MPI_Aint if the attribute was 8 stored with Fortran MPI_xxx_SET_ATTR, and a pointer to int if it was stored with the 9 deprecated Fortran MPI_ATTR_PUT. When an address valued attribute is accessed from 1011 Fortran, then MPI_xxx_GET_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions 12are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may 13 14cause truncation if deprecated attribute functions are used. In C, the deprecated routines 15MPI_Attr_put and MPI_Attr_get behave identical to MPI_Comm_set_attr and 16MPI_Comm_get_attr. 17

Example 16.25

A. Setting an attribute value in C

```
int set_val = 3;
struct foo set_struct;
/* Set a value that is a pointer to an int */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
/* Set a value that is a pointer to a struct */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
/* Set an integer value */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
B. Reading the attribute value in C
```

int flag, *get_val; struct foo *get_struct; /* Upon successful return, get_val == &set_val (and therefore *get_val == 3) */ MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag); /* Upon successful return, get_struct == &set_struct */ MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag); /* Upon successful return, get_val == (void*) 17 */ /* i.e., (MPI_Aint) get_val == 17 */ MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

18

19 20

21

22 23

24 25

26

27

28

29

30 31

32 33

34

35 36

37

38 39

40 41

42

43 44

```
1
     LOGICAL FLAG
\mathbf{2}
     INTEGER IERR, GET_VAL, GET_STRUCT
3
4
     ! Upon successful return, GET_VAL == &set_val, possibly truncated
\mathbf{5}
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
6
     ! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
7
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
8
     ! Upon successful return, GET_VAL == 17
9
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
10
         D. Reading the attribute value with Fortran MPI-2 calls
11
12
     LOGICAL FLAG
13
     INTEGER IERR
14
     INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
15
16
     ! Upon successful return, GET_VAL == &set_val
17
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
18
     ! Upon successful return, GET_STRUCT == &set_struct
19
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
20
     ! Upon successful return, GET_VAL == 17
21
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
22
23
^{24}
     Example 16.26
25
26
         A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
27
     INTEGER IERR, VAL
28
     VAL = 7
29
     CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
30
31
         B. Reading the attribute value in C
32
33
     int flag;
34
     int *value;
35
36
     /* Upon successful return, value points to internal MPI storage and
37
        *value == (int) 7 */
38
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
39
40
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
41
42
     LOGICAL FLAG
43
     INTEGER IERR, VALUE
44
45
     ! Upon successful return, VALUE == 7
46
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
47
48
         D. Reading the attribute value with Fortran MPI-2 calls
```

Unofficial Draft for Comment Only

```
1
LOGICAL FLAG
                                                                                      \mathbf{2}
INTEGER IERR
                                                                                      3
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
                                                                                      4
! Upon successful return, VALUE == 7 (sign extended)
                                                                                      5
                                                                                      6
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      7
                                                                                      8
Example 16.27 A. Setting an attribute value via a Fortran MPI-2 call
                                                                                      9
INTEGER IERR
                                                                                      10
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
                                                                                      11
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
                                                                                     12
VALUE1 = 42
                                                                                     13
VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
                                                                                     14
                                                                                     15
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
                                                                                     16
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
                                                                                     17
                                                                                     18
    B. Reading the attribute value in C
                                                                                     19
int flag;
                                                                                     20
MPI_Aint *value1, *value2;
                                                                                     21
                                                                                     22
/* Upon successful return, value1 points to internal MPI storage and
                                                                                     23
   *value1 == 42 */
                                                                                     24
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
                                                                                     25
/* Upon successful return, value2 points to internal MPI storage and
                                                                                     26
   *value2 == 2^40 */
                                                                                     27
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
                                                                                     28
                                                                                     29
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                     30
                                                                                     31
LOGICAL FLAG
INTEGER IERR, VALUE1, VALUE2
                                                                                     32
                                                                                     33
                                                                                     34
! Upon successful return, VALUE1 == 42
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
                                                                                     35
! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
                                                                                     36
! needed (i.e., the least significant part of the attribute word)
                                                                                     37
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
                                                                                     38
                                                                                     39
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      40
                                                                                     41
LOGICAL FLAG
                                                                                     42
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
                                                                                     43
                                                                                     44
                                                                                     45
! Upon successful return, VALUE1 == 42
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
                                                                                     46
                                                                                     47
! Upon successful return, VALUE2 == 2^40
                                                                                     48
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

¹ The predefined MPI attributes can be integer valued or address valued. Predefined ² integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to ³ the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran,

MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return
 in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD,
 MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound
 for tag value.

⁸ Address valued predefined attributes, such as MPI_WIN_BASE behave as if they were ⁹ put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ¹⁰ ierror) will return in val the base address of the window, converted to an integer. In C, ¹¹ MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window ¹² base, cast to (void *).

 $13 \\ 14$

15

16

17

18 19

20

21

22

23

24

25 26

27

Rationale. The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

Advice to implementors. Implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI_ADDRESS_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI_Attr_put or MPI_Xxx_set_attr), (2) in Fortran with MPI_XXX_SET_ATTR or (3) with the deprecated Fortran routine MPI_ATTR_PUT. Thus, the right choice can be made when the attribute is retrieved. (End of advice to implementors.)

16.3.8 Extra State

28Extra-state should not be modified by the copy or delete callback functions. (This is obvious 29from the C binding, but not obvious from the Fortran binding). However, these functions 30 may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be 31 a pointer to a data structure that is modified by the copy or callback functions; in Fortran, 32 extra-state can be an index into an entry in a COMMON array that is modified by the copy 33 or callback functions. In a multithreaded environment, users should be aware that distinct 34threads may invoke the same callback function concurrently: if this function modifies state 35 associated with extra-state, then mutual exclusion code must be used to protect updates 36 and accesses to the shared state. 37

38 39

16.3.9 Constants

40MPI constants have the same value in all languages, unless specified otherwise. This does not 41 apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, 42etc.) These handles need to be converted, as explained in Section 16.3.4. Constants that 43specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in 44Fortran than C/C++ since in C/C++ the length includes the null terminating character. 45Thus, these constants represent the amount of space which must be allocated to hold the 46largest possible such string, rather than the maximum number of printable characters the 47string could contain. 48

Advice to users. This definition means that it is safe in C/C++ to allocate a buffer to receive a string using a declaration like

char name [MPI_MAX_OBJECT_NAME];

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM must be in Fortran the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better ...) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

16.3.10 Interlanguage Communication

The type matching rules for communication[s] in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 16.28 In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
USE mpi_f08
REAL [ticket250-V.]:: R(5)
INTEGER [ticket250-V.]:: IERR, MYRANK, AOBLEN(1), AOTYPE(1)
[ticket250-V.]TYPE(MPI_Type) :: TYPE
INTEGER (KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: AODISP(1)
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL MPI_TYPE_COMMIT(TYPE, IERR)
CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
```

1

 $\mathbf{2}$

3

45

6 7

8

9

10 11

12

13

14

15

16

17

18

19 20

21

 24

25

26

27

28 29

30

31 32

33

34

35 36

37

38 39

40

41

42

43 44

 $45 \\ 46$

47

```
^{22}_{23} ticket 250-V.
```

```
1
     ELSE
^{2}
         CALL C_ROUTINE(TYPE[ticket250-V.]%MPI_VAL)
3
     END IF
4
5
     /* C code */
6
7
     void C_ROUTINE(MPI_Fint *fhandle)
8
9
      {
10
         MPI_Datatype type;
         MPI_Status status;
11
12
         type = MPI_Type_f2c(*fhandle);
13
14
         MPI_Recv( MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
15
     }
16
17
          MPI implementors may weaken these type matching rules, and allow messages to be
18
     sent with Fortran types and received with C types, and vice versa, when those types match.
19
     I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation
20
      may allow data to be sent with datatype MPI_INTEGER and be received with datatype
21
      MPI_INT. However, such code is not portable.
22
23
24
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

Annex A

Language Bindings Summary

In this section we summarize the specific bindings for C, Fortran, and C++. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

A.1 Defined Values and Handles

A.1.1 Defined Constants

The C and Fortran name is listed in the left column and the C++ name is listed in the middle or right column. Constants with the type **const int** may also be implemented as literal integer constants substituted by the preprocessor.

Return	Codes
C type: const int (or unnamed	enum) C++ type: const int
Fortran type: INTEGER	(or unnamed enum)
MPI_SUCCESS	MPI::SUCCESS
MPI_ERR_BUFFER	MPI::ERR_BUFFER
MPI_ERR_COUNT	MPI::ERR_COUNT
MPI_ERR_TYPE	MPI::ERR_TYPE
MPI_ERR_TAG	MPI::ERR_TAG
MPI_ERR_COMM	MPI::ERR_COMM
MPI_ERR_RANK	MPI::ERR_RANK
MPI_ERR_REQUEST	MPI::ERR_REQUEST
MPI_ERR_ROOT	MPI::ERR_ROOT
MPI_ERR_GROUP	MPI::ERR_GROUP
MPI_ERR_OP	MPI::ERR_OP
MPI_ERR_TOPOLOGY	MPI::ERR_TOPOLOGY
MPI_ERR_DIMS	MPI::ERR_DIMS
MPI_ERR_ARG	MPI::ERR_ARG
MPI_ERR_UNKNOWN	MPI::ERR_UNKNOWN
MPI_ERR_TRUNCATE	MPI::ERR_TRUNCATE
MPI_ERR_OTHER	MPI::ERR_OTHER
MPI_ERR_INTERN	MPI::ERR_INTERN
MPI_ERR_PENDING	MPI::ERR_PENDING
((Continued on next page)

 $\frac{24}{25}$

1	Return Code	s (continued)
2	MPI_ERR_IN_STATUS	MPI::ERR_IN_STATUS
3	MPI_ERR_ACCESS	MPI::ERR_ACCESS
4	MPI_ERR_AMODE	MPI::ERR_AMODE
5	MPI_ERR_ASSERT	MPI::ERR_ASSERT
6	MPI_ERR_BAD_FILE	MPI::ERR_BAD_FILE
7	MPI_ERR_BASE	MPI::ERR_BASE
8	MPI_ERR_CONVERSION	MPI::ERR_CONVERSION
9	MPI_ERR_DISP	MPI::ERR_DISP
10	MPI_ERR_DUP_DATAREP	MPI::ERR_DUP_DATAREP
11	MPI_ERR_FILE_EXISTS	MPI::ERR_FILE_EXISTS
12	MPI_ERR_FILE_IN_USE	MPI::ERR_FILE_IN_USE
13	MPI_ERR_FILE	MPI::ERR_FILE
14	MPI_ERR_INFO_KEY	MPI::ERR_INFO_VALUE
15	MPI_ERR_INFO_NOKEY	MPI::ERR_INFO_NOKEY
16	MPI_ERR_INFO_VALUE	MPI::ERR_INFO_KEY
17	MPI_ERR_INFO	MPI::ERR_INFO
18	MPI_ERR_IO	MPI::ERR_IO
19	MPI_ERR_KEYVAL	MPI::ERR_KEYVAL
20	MPI_ERR_LOCKTYPE	MPI::ERR_LOCKTYPE
21	MPI_ERR_NAME	MPI::ERR_NAME
22	MPI_ERR_NO_MEM	MPI::ERR_NO_MEM
23	MPI_ERR_NOT_SAME	MPI::ERR_NOT_SAME
24	MPI_ERR_NO_SPACE	MPI::ERR_NO_SPACE
25	MPI_ERR_NO_SUCH_FILE	MPI::ERR_NO_SUCH_FILE
26	MPI_ERR_PORT	MPI::ERR_PORT
27	MPI_ERR_QUOTA	MPI::ERR_QUOTA
28	MPI_ERR_READ_ONLY	MPI::ERR_READ_ONLY
29	MPI_ERR_RMA_CONFLICT	MPI::ERR_RMA_CONFLICT
30	MPI_ERR_RMA_SYNC	MPI::ERR_RMA_SYNC
31	MPI_ERR_SERVICE	MPI::ERR_SERVICE
32	MPI_ERR_SIZE	MPI::ERR_SIZE
33	MPI_ERR_SPAWN	MPI::ERR_SPAWN
34	MPI_ERR_UNSUPPORTED_DATAREP	MPI::ERR_UNSUPPORTED_DATAREP
35	MPI_ERR_UNSUPPORTED_OPERATION	MPI::ERR_UNSUPPORTED_OPERATION
36	MPI_ERR_WIN	MPI::ERR_WIN
37	MPI_ERR_LASTCODE	MPI::ERR_LASTCODE
38	[ticket270.]MPI_ERR_RMA_RANGE	[ticket270.]Not defined for C++
39	[ticket270.]MPI_ERR_RMA_ATTACH	[ticket270.]Not defined for C++
ticket 266. 40		
41		
42		
43		
44		
45		
46		

Return Codes for the MPI tool information interface	1
MPI_T_ERR_CANTINIT	2
MPI_T_ERR_NOTINITIALIZED	3
MPI_T_ERR_MEMORY	4
MPI_T_ERR_INVALIDINDEX	5
MPI_T_ERR_INVALIDITEM	6
MPI_T_ERR_INVALIDSESSION	7
MPI_T_ERR_INVALIDHANDLE	8
MPI_T_ERR_OUTOFHANDLES	9
MPI_T_ERR_OUTOFSESSIONS	10
MPI_T_ERR_CVAR_SETNOTNOW	11
MPI_T_ERR_CVAR_SETNEVER	12
MPI_T_ERR_PVAR_NOWRITE	13
MPI_T_ERR_PVAR_NOSTARTSTOP	14
MPI_T_ERR_PVAR_NOATOMIC	15
	16

Buffer Address Constan	ts
C type: void * const	C++ type:
Fortran type: (predefined memory location)	void * const
MPI_BOTTOM	MPI::BOTTOM
MPI_IN_PLACE	MPI::IN_PLACE

C type: const int (or unnamed enum)	C++ type:
Fortran type: INTEGER	const int (or unnamed enum)
MPI_PROC_NULL	MPI::PROC_NULL
MPI_ANY_SOURCE	MPI::ANY_SOURCE
MPI_ANY_TAG	MPI::ANY_TAG
MPI_UNDEFINED	MPI::UNDEFINED
MPI_BSEND_OVERHEAD	MPI::BSEND_OVERHEAD
MPI_KEYVAL_INVALID	MPI::KEYVAL_INVALID
MPI_LOCK_EXCLUSIVE	MPI::LOCK_EXCLUSIVE
MPI_LOCK_SHARED	MPI::LOCK_SHARED
MPI_ROOT	MPI::ROOT

Fortran type: LOGICAL
[ticket234-F.]MPI_SUBARRAYS_SUPPORTED (Fortran only)
[ticket238-J.][ticket229.1.]MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)

 $\frac{41}{42}$

(14	E .		
		Status size and reserve	ed index values (Fortran only)
		Fortran type: INTEGER	
		÷ -	ot defined for C++
			bt defined for $C++$
			ot defined for C++
		_	ot defined for C++
		Variable Addre	ess Size (Fortran only)
		Fortran type: INTEGER	· · · · · · · · · · · · · · · · · · ·
		MPI_ADDRESS_KIND	Not defined for C++
		[ticket265.]MPI_COUNT_KIN	ID [ticket265.]Not defined for C++
		MPI_INTEGER_KIND	Not defined for C++
		MPI_OFFSET_KIND	Not defined for C++
		Error-ha	ndling specifiers
	C typ	De: MPI_Errhandler	C++ type: MPI::Errhandler
	÷ -	an type: INTEGER	
	tick	et231-C.]or TYPE(MPI_Errhand	ler)
	MPI	ERRORS_ARE_FATAL	MPI::ERRORS_ARE_FATAL
	1011 1		MIFIERRORS_ARE_FATAL
			MPI::ERRORS_RETURN
		ERRORS_RETURN	MPI::ERRORS_RETURN
			MPI::ERRORS_RETURN
		ERRORS_RETURN	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS
	MPI_	ERRORS_RETURN	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS
	MPI	ERRORS_RETURN Maximum const int (or unnamed enum)	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS
	MPI_ C type: Fortran	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS
	MPI_ C type: Fortran MPI_M	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS A Sizes for Strings C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAM
	MPI_ C type: Fortran MPI_M. [ticket2	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 204.]MPI_MAX_LIBRARY_VERS	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS A Sizes for Strings C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 204.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI MPI::MAX_ERROR_STRING
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_DATAREP_STRING	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS a Sizes for Strings C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_DATAREP_STRING
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_INFO_KEY	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS A Sizes for Strings C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAME SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_INFO_KEY AX_INFO_VAL	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_INFO_KEY AX_INFO_VAL	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME
	MPI_ C type: Fortran MPI_M. [ticket2 MPI_M. MPI_M. MPI_M. MPI_M.	ERRORS_RETURN Maximum const int (or unnamed enum) type: INTEGER AX_PROCESSOR_NAME 04.]MPI_MAX_LIBRARY_VERS AX_ERROR_STRING AX_ERROR_STRING AX_DATAREP_STRING AX_INFO_KEY AX_INFO_VAL AX_OBJECT_NAME	MPI::ERRORS_RETURN MPI::ERRORS_THROW_EXCEPTIONS C++ type: const int (or unnamed enum) MPI::MAX_PROCESSOR_NAMI SION_STRING MPI::MAX_ERROR_STRING MPI::MAX_INFO_KEY MPI::MAX_INFO_VAL MPI::MAX_OBJECT_NAME

Named Predet	ined Datatypes	$C/C++$ types $_2$
C type: MPI_Datatype	C++ type: MPI::Datatype	3
Fortran type: INTEGER		4
[ticket231-C.]or TYPE(MPI_Datatype)		5
MPI_CHAR	MPI::CHAR	char 6
		(treated as printable
		character) 8
MPI_SHORT	MPI::SHORT	signed short int
 MPI_INT	MPI::INT	signed int 10
_ MPI_LONG	MPI::LONG	signed long 11
MPI_LONG_LONG_INT	MPI::LONG_LONG_INT	signed long long
MPI_LONG_LONG	MPI::LONG_LONG	long long (synonym)
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR	signed char 14
		(treated as integral value
MPI_UNSIGNED_CHAR	MPI::UNSIGNED_CHAR	unsigned char ¹⁶
		(treated as integral value
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	unsigned short 18
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	unsigned int 19
MPI_UNSIGNED_LONG		
	MPI::UNSIGNED_LONG	unsigned long 20
MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	MPI::FLOAT	float 22
MPI_DOUBLE	MPI::DOUBLE	double 23
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE	long double 24
MPI_WCHAR	MPI::WCHAR	wchar_t 25
		(defined in <stddef.h>)</stddef.h>
		(treated as printable
		character) 28
MPI_C_BOOL	(use C datatype handle)	_Bool 29
MPI_INT8_T	(use C datatype handle)	int8_t 30
MPI_INT16_T	(use C datatype handle)	int16_t 31
MPI_INT32_T	(use C datatype handle)	int32_t 32
MPI_INT64_T	(use C datatype handle)	int64_t 33
MPI_UINT8_T	(use C datatype handle)	uint8_t 34
MPI_UINT16_T	(use C datatype handle)	uint16_t 35
MPI_UINT32_T	(use C datatype handle)	uint32_t 36
MPI_UINT64_T	(use C datatype handle)	uint64_t 37
MPI_AINT	(use C datatype handle)	MPI_Aint 38
[ticket265.]MPI_COUNT	[ticket265.](use C datatype handle)	[ticket265.]MPI_Count
MPI_OFFSET	(use C datatype handle)	MPI_Offset 40
MPI_C_COMPLEX	(use C datatype handle)	float _Complex 41
MPI_C_FLOAT_COMPLEX	(use C datatype handle)	float _Complex 42
MPI_C_DOUBLE_COMPLEX	(use C datatype handle)	double _Complex43
MPI_C_LONG_DOUBLE_COMPLEX	(use C datatype handle)	long double _Complex
MPI_BYTE	MPI::BYTE	(any C/C++ type)

Unofficial Draft for Comment Only

Named Predefit C type: MPI_Datatype	C++ type: MPI::E	atatype	Fortran types
Fortran type: INTEGER			
[ticket231-C.]or TYPE(MPI_Dataty	rpe)		
MPI_INTEGER	MPI::INTEGER		INTEGER
 MPI_REAL	MPI::REAL		REAL
MPI_DOUBLE_PRECISION	MPI::DOUBLE_PR	ECISION	DOUBLE PRECISION
MPI_COMPLEX	MPI::F_COMPLEX		COMPLEX
MPI_LOGICAL	MPI::LOGICAL		LOGICAL
MPI_CHARACTER	MPI::CHARACTER		CHARACTER(1)
MPI_AINT	(use C datatype l	handle)	INTEGER (KIND=MPI_ADDRESS_KIN
[ticket265.]MPI_COUNT	(use C datatype]		[ticket265.]INTEGER (KIND=MPI_CO
MPI_OFFSET	(use C datatype]		INTEGER (KIND=MPI_OFFSET_KIND)
MPI_BYTE	MPI::BYTE	/	(any Fortran type)
MPI_PACKED	MPI::PACKED		(any Fortran type)
MPI::DOUBLE_COMPLE		-	x <double></double>
MPI::LONG_DOUBLE_CO		-	x <long double=""></long>
MPI::LONG_DOUBLE_CO	OMPLEX Types (Fortran)	Complex	
MPI::LONG_DOUBLE_CO Optional datat	OMPLEX	Complex	<pre>x<long double=""></long></pre>
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER	OMPLEX types (Fortran) C++ type: MPI::E	Complex	<pre>x<long double=""></long></pre>
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty	OMPLEX types (Fortran) C++ type: MPI::E	Complex	<pre>x<long double=""> Fortran types</long></pre>
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX	OMPLEX types (Fortran) C++ type: MPI::E Type) MPI::F_DOUBLE_C	Complex	Fortran types
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1	OMPLEX types (Fortran) C++ type: MPI::E Type) MPI::F_DOUBLE_C MPI::INTEGER1	Complex	Fortran types DOUBLE COMPLEX INTEGER*1
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2	OMPLEX types (Fortran) C++ type: MPI::E mpe) MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2	Complex	Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2
Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4	OMPLEX types (Fortran) C++ type: MPI::E mpe) MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4	Complex	Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8	OMPLEX types (Fortran) C++ type: MPI::E mpe) MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2	Complex	Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4 INTEGER*8
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16	OMPLEX types (Fortran) C++ type: MPI::E mpe) MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8	Complex	Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*{ticket231-C.][8]2 INTEGER*8 INTEGER*8 INTEGER*16
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER1 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2	OMPLEX types (Fortran) C++ type: MPI::I mpe) MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2	Complex	Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*8 INTEGER*8 INTEGER*16 REAL*2
Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4	OMPLEX Eypes (Fortran) C++ type: MPI::E MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2 MPI::REAL4	Complex	Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*8 INTEGER*8 INTEGER*16 REAL*2 REAL*4
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL8	OMPLEX types (Fortran) C++ type: MPI::I mpe) MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2	Complex	Fortran types Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8
Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16	OMPLEX Eypes (Fortran) C++ type: MPI::E MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2 MPI::REAL4	Complex	Fortran types Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4	OMPLEX Eypes (Fortran) C++ type: MPI::E MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2 MPI::REAL4	Complex	Fortran types Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty] MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4 MPI_COMPLEX8	OMPLEX Eypes (Fortran) C++ type: MPI::E MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2 MPI::REAL4	Complex	Fortran types Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*2 REAL*4 REAL*8 REAL*4 COMPLEX*4 COMPLEX*8
MPI::LONG_DOUBLE_CO Optional datat C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Dataty MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL4 MPI_REAL4 MPI_REAL4 MPI_REAL4	OMPLEX Eypes (Fortran) C++ type: MPI::E MPI::F_DOUBLE_C MPI::INTEGER1 MPI::INTEGER2 MPI::INTEGER4 MPI::INTEGER8 MPI::REAL2 MPI::REAL4	Complex	Fortran types Fortran types DOUBLE COMPLEX INTEGER*1 INTEGER*[ticket231-C.][8]2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4

Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Datatype) MPI_FLOAT_INT MPI::FLOAT_INT MPI_DOUBLE_INT MPI::DOUBLE_INT MPI_LONG_INT MPI::TWOINT MPI_SHORT_INT MPI::SHORT_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_Datatypes for reduction functions (Fortran) type: MPI_Datatype type: MPI_Datatype C++ type: MPI::Datatype PI_2REAL MPI::TWOREAL PI_2DOUBLE_PRECISION MPI::TWOINTEGER PI_2INTEGER MPI::TWOINTEGER C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype Fortran type: INTEGER MPI::UB MPI_UB MPI::UB	C type: MPI_Datatype	on functions (C and C++) C++ type: MPI::Datatype
MPI_FLOAT_INT MPI:FLOAT_INT MPI_DOUBLE_INT MPI:DOUBLE_INT MPI_LONG_INT MPI::DOUGLE_INT MPI_2INT MPI::TWOINT MPI_SHORT_INT MPI::SHORT_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::Datatype C++ type: MPI::Datatype rtran type: INTEGER cket231-C.]or TYPE(MPI_Datatype) MPI::TWOREAL PI_2DOUBLE_PRECISION MPI::TWOINTEGER C type: MPI_Datatype C++ type: MPI::Datatype C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER MPI::UB MPI_LB MPI::UB MPI::UB MPI_LB MPI::UB MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_SELF MPI::COMM_SELF MPI_COMM_SELF MPI::COMM_SELF MPI::COMM_SELF	Fortran type: INTEGER	
MPI_FLOAT_INT MPI:FLOAT_INT MPI_DOUBLE_INT MPI:DOUBLE_INT MPI_LONG_INT MPI::DOUGLE_INT MPI_2INT MPI::TWOINT MPI_SHORT_INT MPI::SHORT_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_CONG_DOUBLE_INT MPI::Datatype C++ type: MPI::Datatype rtran type: INTEGER cket231-C.]or TYPE(MPI_Datatype) MPI::TWOREAL PI_2DOUBLE_PRECISION MPI::TWOINTEGER C type: MPI_Datatype C++ type: MPI::Datatype C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER MPI::UB MPI_LB MPI::UB MPI::UB MPI_LB MPI::UB MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_SELF MPI::COMM_SELF MPI_COMM_SELF MPI::COMM_SELF MPI::COMM_SELF		atype)
MPI_LONG_INT MPI:LONG_INT MPI_2INT MPI:TWOINT MPI_SHORT_INT MPI:SHORT_INT MPI_LONG_DOUBLE_INT MPI:CONG_DOUBLE_INT Datatypes for reduction functions (Fortran) type: MPI_Datatype C++ type: MPI::Datatype tran type: INTEGER C++ type: MPI::Datatype cket231-C.]or TYPE(MPI_Datatype) MPI::TWOREAL PI_2REAL MPI::TWOODUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_LB MPI::LB MPI_LB MPI::LB MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI.COMM_SELF MPI::COMM_SELF MPI:COMM_SELF MPI::COMM_SELF MPI::COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER	MPI_FLOAT_INT	
MPI_2INT MPI::TWOINT MPI_SHORT_INT MPI::SHORT_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT Datatypes for reduction functions (Fortran) type: MPI_Datatype C++ type: MPI::Datatype rtran type: INTEGER C++ type: MPI::Datatype eket231-C.]or TYPE(MPI_Datatype) MPI::TWOREAL PI_2REAL MPI::TWOOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype ticket231-C.]or TYPE(MPI_Datatype) MPI::UB MPI_UB MPI::UB MPI_LB MPI::LB MPI_LB MPI::LB MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER	MPI_DOUBLE_INT	MPI::DOUBLE_INT
MPI_SHORT_INT MPI::SHORT_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_LONG_DOUBLE_INT MPI::CONG_DOUBLE_INT MPI_type C++ type: MPI::Datatype rtran type: INTEGER C++ type: MPI::Datatype cket231-C.]or TYPE(MPI_Datatype) MPI::TWOREAL PI_2REAL MPI::TWOODUBLE_PRECISION PI_2DOUBLE_PRECISION MPI::TWOINTEGER Ctype: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype Fortran type: INTEGER MPI::UB MPI_UB MPI::UB MPI_LB MPI::UB MPI_LB MPI::LB MPI_LB MPI::CB MPI_COMM_CORM C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI::COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unmaned enum) Fortran type: INTEGER Fortran type: INTEGER	MPI_LONG_INT	MPI::LONG_INT
MPI_LONG_DOUBLE_INT MPI::LONG_DOUBLE_INT Datatypes for reduction functions (Fortran) type: MPI_Datatype C++ type: MPI::Datatype rtran type: INTEGER Cheetee MPI::TWOREAL PI_2REAL MPI::TWODUBLE_PRECISION PI_2DOUBLE_PRECISION MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_UB MPI::UB MPI_LB MPI::UB MPI_LB MPI::CB C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER	MPI_2INT	MPI::TWOINT
Datatypes for reduction functions (Fortran) type: MPI_Datatype C++ type: MPI::Datatype rtran type: INTEGER cket231-C.]or TYPE(MPI_Datatype) PI_2REAL MPI::TWOREAL PI_2DOUBLE_PRECISION MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype Fortran type: INTEGER MPI::UB MPI_UB MPI::UB MPI_LB MPI::LB MPI_LB MPI::LB MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER	MPI_SHORT_INT	MPI::SHORT_INT
type: MPI_Datatype C++ type: MPI::Datatype rtran type: INTEGER Cket231-C.]or TYPE(MPI_Datatype) PI_2REAL MPI::TWOREAL PI_2DUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype ticket231-C.]or TYPE(MPI_Datatype) MPI::UB MPI_UB MPI::UB MPI_LB MPI::UB MPI_LB MPI::COMM_MORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER	MPI_LONG_DOUBLE_INT	MPI::LONG_DOUBLE_INT
type: MPI_Datatype C++ type: MPI::Datatype rtran type: INTEGER Cket231-C.]or TYPE(MPI_Datatype) PI_2REAL MPI::TWOREAL PI_2DUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype ticket231-C.]or TYPE(MPI_Datatype) MPI::UB MPI_UB MPI::UB MPI_LB MPI::UB MPI_LB MPI::COMM_MORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER		
type: MPI_Datatype C++ type: MPI::Datatype rtran type: INTEGER Cket231-C.]or TYPE(MPI_Datatype) PI_2REAL MPI::TWOREAL PI_2DUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype ticket231-C.]or TYPE(MPI_Datatype) MPI::UB MPI_UB MPI::UB MPI_LB MPI::UB MPI_LB MPI::COMM_MORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER		
rtran type: INTEGER cket231-C.]or TYPE(MPI_Datatype) PI_2REAL MPI::TWOREAL PI_2DUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_LB MPI::LB MPI_LB MPI::LB Reserved communicators C type: MPI_Comm C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER		. ,
cket231-C.]or TYPE(MPI_Datatype) PI_2REAL MPI::TWOREAL PI_2DOUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_LB MPI::LB MPI_LB MPI::LB MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER	• - • -	C++ type: MPI::Datatype
PI_2REAL MPI::TWOREAL PI_2DOUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER C++ type: MPI::Datatype MPI_UB MPI::UB MPI_LB MPI::LB MPI_LB MPI::LB C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_LB MPI::LB MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_SELF MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER	• =	
PI_2DOUBLE_PRECISION MPI::TWODOUBLE_PRECISION PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_LB MPI::LB C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER	· · · · · · · · · · · · · · · · · · ·	-
PI_2INTEGER MPI::TWOINTEGER Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER MPI::UB ticket231-C.]or TYPE(MPI_Datatype) MPI::UB MPI_UB MPI::LB MPI_LB MPI::LB Reserved communicators C C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER C type: const int (or unnamed enum) Fortran type: INTEGER	_	
Special datatypes for constructing derived datatypes C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_LB MPI::LB C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER		
C type: MPI_Datatype C++ type: MPI::Datatype Fortran type: INTEGER ticket231-C.]or TYPE(MPI_Datatype) MPI_UB MPI::UB MPI_LB MPI::LB C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF C type: const int (or unnamed enum) Fortran type: INTEGER	MPI_2INTEGER	MPI:: I WOIN LEGER
MPI_LB MPI::LB Reserved communicators C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER	• - • -	C++ type: MP1::Datatype
Reserved communicators C type: MPI_Comm C++ type: MPI::Intracomm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data	type)
Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER	type) MPI::UB
[ticket231-C.]or TYPE(MPI_Comm) MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB Reserved c	type) MPI::UB MPI::LB
MPI_COMM_WORLD MPI::COMM_WORLD MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB Reserved c C type: MPI_Comm	type) MPI::UB MPI::LB
MPI_COMM_SELF MPI::COMM_SELF Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm
Communicator split type constants C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm
C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm MPI_COMM_WORLD	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD
C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm MPI_COMM_WORLD	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD
C type: const int (or unnamed enum) Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm MPI_COMM_WORLD	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD
Fortran type: INTEGER	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm MPI_COMM_WORLD MPI_COMM_SELF	<pre>type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF</pre>
· -	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Co MPI_COMM_WORLD MPI_COMM_SELF	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Commission MPI_COMM_WORLD MPI_COMM_SELF C type: const int	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum)
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Command MPI_COMM_WORLD MPI_COMM_SELF C type: const int Fortran type: INTEG	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum) SER
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Command MPI_COMM_WORLD MPI_COMM_SELF C type: const int Fortran type: INTEG	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum) SER
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Command MPI_COMM_WORLD MPI_COMM_SELF C type: const int Fortran type: INTEG	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum) SER
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Command MPI_COMM_WORLD MPI_COMM_SELF C type: const int Fortran type: INTEG	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum) SER
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Command MPI_COMM_WORLD MPI_COMM_SELF C type: const int Fortran type: INTEG	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum) SER
	Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Data MPI_UB MPI_LB C type: MPI_Comm Fortran type: INTEGER [ticket231-C.]or TYPE(MPI_Comm MPI_COMM_WORLD MPI_COMM_SELF C type: const int Fortran type: INTEG	type) MPI::UB MPI::LB communicators C++ type: MPI::Intracomm mm) MPI::COMM_WORLD MPI::COMM_SELF split type constants (or unnamed enum) SER

	Results of communicate	or and g	group comparisons
	C type: const int (or unname	ed enum)	C++ type: const int
	Fortran type: INTEGER		(or unnamed enum)
	MPI_IDENT		MPI::IDENT
	MPI_CONGRUENT		MPI::CONGRUENT
	MPI_SIMILAR		MPI::SIMILAR
	MPI_UNEQUAL		MPI::UNEQUAL
	Environment	al inqui	ry keys
	C type: const int (or unnamed	l enum)	C++ type: const int
	Fortran type: INTEGER		(or unnamed enum)
	MPI_TAG_UB		MPI::TAG_UB
	MPI_IO		MPI::IO
	 MPI_HOST		MPI::HOST
	MPI_WTIME_IS_GLOBAL		MPI::WTIME_IS_GLOBAL
	Collective	Operat	tions
_	C type: MPI_Op	-	De: const MPI::Op
	Fortran type: INTEGER	C I I UY	e. const in 1op
	[ticket231-C.]or TYPE(MPI_Op)		
-	MPI_MAX	MPI::MA	×
		MPI::MI	
	MPI_MIN		
	MPI_SUM	MPI::SU	
	MPI_PROD	MPI::PR	
	MPI_MAXLOC	MPI::MA	
	MPI_MINLOC	MPI::MI	
	MPI_BAND	MPI::BA	
	MPI_BOR	MPI::BO	
	MPI_BXOR	MPI::BX	
	MPI_LAND	MPI::LAI	
	MPI_LOR	MPI::LO	P
	MPI_LXOR	MPI::LX	OR
	MPI_LXOR MPI_REPLACE	MPI::LX MPI::RE	DR PLACE
	MPI_LXOR	MPI::LX MPI::RE	OR
	MPI_LXOR MPI_REPLACE	MPI::LX MPI::RE	DR PLACE

Null Ha	andles
/Fortran name	C++ name
C type / Fortran type	C++ type
PI_GROUP_NULL	MPI::GROUP_NULL
MPI_Group / INTEGER	const MPI::Group
ket231-C.] or TYPE(MPI_Group)	_
I_COMM_NULL	MPI::COMM_NULL
MPI_Comm / INTEGER	1)
cket231-C.] or TYPE(MPI_Comm)	
PI_DATATYPE_NULL	MPI::DATATYPE_NULL
MPI_Datatype / INTEGER	const MPI::Datatype
cket231-C.] or TYPE(MPI_Datatype)	
PI_REQUEST_NULL	MPI::REQUEST_NULL
$\texttt{MPI}_{\texttt{Request}} / \texttt{INTEGER}$	const MPI::Request
cket231-C.] or TYPE(MPI_Request)	-
PI_OP_NULL	MPI::OP_NULL
MPI_Op / INTEGER	const MPI::Op
icket231-C.] or TYPE(MPI_Op)	-
PI_ERRHANDLER_NULL	MPI::ERRHANDLER_NULL
MPI_Errhandler / INTEGER	const MPI::Errhandler
cket231-C.] or TYPE(MPI_Errhandle	r)
PI_FILE_NULL	MPI::FILE_NULL
MPI_File / INTEGER	
icket231-C.] or TYPE(MPI_File)	
PI_INFO_NULL	MPI::INFO_NULL
MPI_Info / INTEGER	const MPI::Info
cket231-C.] or TYPE(MPI_Info)	
PI_WIN_NULL	MPI::WIN_NULL
MPI_Win / INTEGER	
cket231-C.] or TYPE(MPI_Win)	
cket274.]MPI_MESSAGE_NULL	[ticket274.]Not defined for C++
icket274.] MPI_Message / INTEGER	L 3
cket231-C.] or TYPE(MPI_Message)	
C++ type: See Section 16.1.7 on pa	age 636 regarding
class hierarchy and the specific type	
Empty	group
C type: MPI_Group	C++ type: const MPI::Group
Fortran type: INTEGER	· - 1
[ticket231-C.]or TYPE(MPI_Group)	
MPI_GROUP_EMPTY	MPI::GROUP_EMPTY

Null Handles

1	Topologies	
2	C type: const int (or unnamed enum)	C++ type: const int
3	Fortran type: INTEGER	(or unnamed enum)
4	MPI_GRAPH	MPI::GRAPH
5	MPI_CART	MPI::CART
6	MPI_DIST_GRAPH	MPI::DIST_GRAPH
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

		1
Predefined functions C/Fortran name	C++ name	2
C type / Fortran type [ticket230-B.]with mpi module	C + + type	4
[ticket230-B.]/ Fortran type with mpi_f08 module	C + + type	5
MPI_COMM_NULL_COPY_FN	MPI_COMM_NULL_COP	
MPI_Comm_copy_attr_function	same as in C ^{1})	7
/ COMM_COPY_ATTR_[ticket250-V.][FN]FUNCTION	same as m o)	8
/ [ticket230-B.]PROCEDURE(MPI_Comm_copy_attr_function) ²)		9
MPI_COMM_DUP_FN	MPI_COMM_DUP_FN	10
MPI_Comm_copy_attr_function	same as in C 1)	11
/ COMM_COPY_ATTR_[ticket250-V.][FN]FUNCTION		12
/ [ticket230-B.]PROCEDURE(MPI_Comm_copy_attr_function) ²)		13
MPI_COMM_NULL_DELETE_FN	MPI_COMM_NULL_DEL	
MPI_Comm_delete_attr_function	same as in C 1)	15
/ COMM_DELETE_ATTR_[ticket250-V.][FN]FUNCTION		16
<pre>/ [ticket230-B.]PROCEDURE(MPI_Comm_delete_attr_function) ²)</pre>		17
MPI_WIN_NULL_COPY_FN	MPI_WIN_NULL_COPY_	
MPI_Win_copy_attr_function	same as in C ^{1})	19
/ WIN_COPY_ATTR_[ticket250-V.] [FN] FUNCTION		20
<pre>/ [ticket230-B.]PROCEDURE(MPI_Win_copy_attr_function)²)</pre>		21
MPI_WIN_DUP_FN	MPI_WIN_DUP_FN	22
MPI_Win_copy_attr_function	same as in C ^{1})	23
/ WIN_COPY_ATTR_[ticket250-V.][FN]FUNCTION		24
<pre>/ [ticket230-B.]PROCEDURE(MPI_Win_copy_attr_function)²)</pre>		25
MPI_WIN_NULL_DELETE_FN	MPI_WIN_NULL_DELET	EE€N
MPI_Win_delete_attr_function	same as in C ^{1})	27
/ WIN_DELETE_ATTR_[ticket250-V.][FN]FUNCTION		28
<pre>/ [ticket230-B.]PROCEDURE(MPI_Win_delete_attr_function) ²)</pre>		29
MPI_TYPE_NULL_COPY_FN	MPI_TYPE_NULL_COPY	∕ F\$N
MPI_Type_copy_attr_function	same as in C ^{1})	31
/ TYPE_COPY_ATTR_[ticket250-V.][FN]FUNCTION		32
<pre>/ [ticket230-B.]PROCEDURE(MPI_Type_copy_attr_function) ²)</pre>		33
MPI_TYPE_DUP_FN	MPI_TYPE_DUP_FN	34
MPI_Type_copy_attr_function	same as in C^{1}	35
/ TYPE_COPY_ATTR_[ticket250-V.][FN]FUNCTION	· · · · · · · · · · · · · · · · · · ·	36
<pre>/ [ticket230-B.]PROCEDURE(MPI_Type_copy_attr_function)²)</pre>		37
MPI_TYPE_NULL_DELETE_FN	MPI_TYPE_NULL_DELE	TE₃€N
MPI_Type_delete_attr_function	same as in C^{1}	39
/ TYPE_DELETE_ATTR_[ticket250-V.][FN]FUNCTION	,	40
/ [ticket230-B.]PROCEDURE(MPI_Type_delete_attr_function) ²)		41
¹ See the advice to implementors [ticket230-B.](on page 281) an	d advice to users (on page	2813
on [ticket230-B.]the predefined C functions MPI_COMM_NU		43
Section 6.7.2 on page 278	,	44
$[ticket 230-B]^2$ See the advice to implementors (on page 281) an	d advice to users (on page	282)
[ticket230-B.] on the predefined Fortran functions MPI_COM	· · · · · · · · · · · · · · · · · · ·	46
	,	
[ticket230-B.] Section 6.7.2 on page 278		47

1	Deprecated predef	ined functions
2	C/Fortran name	C++ name
3	C type / Fortran type	C++ type
4	MPI_NULL_COPY_FN	MPI::NULL_COPY_FN
5	MPI_Copy_function / COPY_FUNCTIO	MPI::Copy_function
6	MPI_DUP_FN	MPI::DUP_FN
7	MPI_Copy_function / COPY_FUNCTIO	MPI::Copy_function
8	MPI_NULL_DELETE_FN	MPI::NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUN	
10	/	
11		
12	Predefined Attr	ibute Keys
13	C type: const int (or unnamed enum)	C++ type:
14	Fortran type: INTEGER	const int (or unnamed enum)
15	MPI_APPNUM	MPI::APPNUM
16	MPI_LASTUSEDCODE	MPI::LASTUSEDCODE
17	MPI_UNIVERSE_SIZE	MPI::UNIVERSE_SIZE
18	MPI_WIN_BASE	MPI::WIN_BASE
19	MPI_WIN_DISP_UNIT	MPI::WIN_DISP_UNIT
20	MPI_WIN_SIZE	MPI::WIN_SIZE
21	[ticket270.]MPI_WIN_CREATE_FLAVOR	[ticket270.]Not defined for C++
22	[ticket270.]MPI_WIN_MODEL	[ticket270.]Not defined for C++
23		
ticlet 270 ²⁴		
ticket270. $_{25}^{-1}$		
26	MPI Window Crea	te Flavors
27	C type: const int (or	
28	Fortran type: INTEGER	······································
29	MPI_WIN_FLAVOR_CR	FATE
30	MPI_WIN_FLAVOR_AL	
31	MPI_WIN_FLAVOR_DY	
32	[ticket284.]MPI_WIN_F	
33		
34		
ticket270. $_{35}$		
36	MPI Window Mod	lels
37	C type: const int (or	
38	Fortran type: INTEGER	· · · · · · · · · · · · · · · · · · ·
	MPI_WIN_SEPARATE	,
39		
39 40 41	MPI_WIN_UNIFIED	
40		
40 41 42		
40 41 42 43		
40 41 42 43 44		
40 41 42 43 44 45		
40 41 42 43 44 45 46		
40 41 42 43 44 45		

Mode Col	istantis	-
C type: const int (or unnamed enum)	C++ type:	2
Fortran type: INTEGER	<pre>const int (or unnamed enum)</pre>	3
MPI_MODE_APPEND	MPI::MODE_APPEND	4
MPI_MODE_CREATE	MPI::MODE_CREATE	5
MPI_MODE_DELETE_ON_CLOSE	MPI::MODE_DELETE_ON_CLOSE	6
MPI_MODE_EXCL	MPI::MODE_EXCL	7
MPI_MODE_NOCHECK	MPI::MODE_NOCHECK	8
MPI_MODE_NOPRECEDE	MPI::MODE_NOPRECEDE	9
MPI_MODE_NOPUT	MPI::MODE_NOPUT	10
MPI_MODE_NOSTORE	MPI::MODE_NOSTORE	11
MPI_MODE_NOSUCCEED	MPI::MODE_NOSUCCEED	12
MPI_MODE_RDONLY	MPI::MODE_RDONLY	13
MPI_MODE_RDWR	MPI::MODE_RDWR	14
MPI_MODE_SEQUENTIAL	MPI::MODE_SEQUENTIAL	15
MPI_MODE_UNIQUE_OPEN	MPI::MODE_UNIQUE_OPEN	16
MPI_MODE_WRONLY	MPI::MODE_WRONLY	17
		18
		19

Mode Constants

Datatype Decoding Constants

Datatype Decoding	Constants
C type: const int (or unnamed enum)	C++ type:
Fortran type: INTEGER	<pre>const int (or unnamed enum)</pre>
MPI_COMBINER_CONTIGUOUS	MPI::COMBINER_CONTIGUOUS
MPI_COMBINER_DARRAY	MPI::COMBINER_DARRAY
MPI_COMBINER_DUP	MPI::COMBINER_DUP
MPI_COMBINER_F90_COMPLEX	MPI::COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI::COMBINER_F90_INTEGER
MPI_COMBINER_F90_REAL	MPI::COMBINER_F90_REAL
MPI_COMBINER_HINDEXED_INTEGER	MPI::COMBINER_HINDEXED_INTEGER
MPI_COMBINER_HINDEXED	MPI::COMBINER_HINDEXED
MPI_COMBINER_HVECTOR_INTEGER	MPI::COMBINER_HVECTOR_INTEGER
MPI_COMBINER_HVECTOR	MPI::COMBINER_HVECTOR
MPI_COMBINER_INDEXED_BLOCK	MPI::COMBINER_INDEXED_BLOCK
[ticket280.]MPI_COMBINER_HINDEXED_BLOCK	
MPI_COMBINER_INDEXED	MPI::COMBINER_INDEXED
MPI_COMBINER_NAMED	MPI::COMBINER_NAMED
MPI_COMBINER_RESIZED	MPI::COMBINER_RESIZED
MPI_COMBINER_STRUCT_INTEGER	MPI::COMBINER_STRUCT_INTEGER
MPI_COMBINER_STRUCT	MPI::COMBINER_STRUCT
MPI_COMBINER_SUBARRAY	MPI::COMBINER_SUBARRAY
MPI_COMBINER_VECTOR	MPI::COMBINER_VECTOR

C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_THREAD_FUNNELED MPI:THREAD_FUNNELED MPI_THREAD_MULTIPLE MPI:THREAD_SERIALIZED MPI_THREAD_SINGLE MPI:THREAD_SINGLE File Operation Constants, Part 1 Type: const MPI_Dffset (or unnamed enum) C++ type: ortran type: INTEGER CHIMD-MPI_OFFSET_KIND) const MPI::0ffset (or unnamed enum) File Operation Constants, Part 1 The Operation Constants, Part 1 Thread CHIMD-MPI_OFFSET_KIND) const MPI_:Dffset (or unnamed enum) C++ type: ortran type: INTEGER C type: const int (or unnamed enum) C++ type: ortran type: INTEGER const int (or unnamed enum) MPI::DISTRIBUTE_DCURE MPI::DISTRIBUTE_DELC MPI::DISTRIBUTE_DELCK MPI_DISTRIBUTE_NONE MPI::DISTRIBUTE_NONE MPI_DISTRIBUTE_NONE MPI::DISTRIBUTE_NONE MPI_ORDER_C MPI::ORDER_CCR MPI_:DISTRIBUTE_NONE MPI::DISTRIBUTE_NONE<	Threads Con	stants		
MPI_THREAD_FUNNELED MPI::THREAD_FUNNELED MPI_THREAD_MULTIPLE MPI::THREAD_SERIALIZED MPI_THREAD_SERIALIZED MPI::THREAD_SERIALIZED MPI_THREAD_SINGLE MPI::THREAD_SINGLE File Operation Constants, Part 1 C type: const MPI_Offset (or unnamed enum) C++ type: ortran type: INTEGER (KIND=MPI_OFFSET_KIND) C type: const int (or unnamed enum) MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CURRENT_CONSTINUTE_BLOCK MPI:DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_CYCLIC MPI::DISTRIBUTE_DFLT_DARG MPI_DISTRIBUTE_DFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_ORDER_C MPI::DISTRIBUTE_NONE MPI_ORDER_C MPI::DISTRIBUTE_NONE MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_SET MPI::SEEK_SET <td <="" colspan="2" th=""><th>C type: const int (or unnamed enum)</th><th>C++ type:</th></td>	<th>C type: const int (or unnamed enum)</th> <th>C++ type:</th>		C type: const int (or unnamed enum)	C++ type:
MPI_THREAD_MULTIPLE MPI::THREAD_SERIALIZED MPI_THREAD_SERIALIZED MPI::THREAD_SERIALIZED MPI_THREAD_SINGLE MPI::THREAD_SINGLE File Operation Constants, Part 1 Ctype: const MPI_OFFSet (or unnamed enum) C++ type: const MPI_CURRENT MPI::DISPLACEMENT_CURRENT File Operation Constants, Part 2 C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_DFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_ORDER_C MPI::ORDER_C MPI_ORDER_C MPI::ORDER_C MPI_ORDER_C MPI::ORDER_C MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_SEEK_SET MPI::SEEK_SET MPI_SEEK_SET MPI::SEEK_SET MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER	Fortran type: INTEGER	const int (or unnamed enum)		
MPI_THREAD_SERIALIZED MPI::THREAD_SERIALIZED MPI_THREAD_SINGLE MPI::THREAD_SINGLE File Operation Constants, Part 1 Stype: const MPI_Offset (or unnamed enum) C++ type: ortran type: INTEGER (KIND=MPI_OFFSET_KIND) const MPI::Offset (or unnamed enum) MPI::DISPLACEMENT_CURRENT File Operation Constants, Part 2 C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_CYCLIC MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_CYCLIC MPI::DISTRIBUTE_DFLT_DARG MPI_DISTRIBUTE_OFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_ORDER_C MPI::ORDER_C MPI_ORDER_FORTRAN MPI::DISTRIBUTE_NONE MPI_ORDER_FORTRAN MPI::SEEK_CUR MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_SEEK_SET MPI::SEEK_SET MPI::SEEK_SET	MPI_THREAD_FUNNELED	MPI::THREAD_FUNNELED		
MPI_THREAD_SINGLE MPI::THREAD_SINGLE File Operation Constants, Part 1 Ctype: const MPI_Offset (or unnamed enum) C++ type: fortran type: INTEGER (KIND=MPI_OFFSET_KIND) const MPI::Offset (or unnamed enum) MPI_DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI_DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI:DISTRIBUTE_CYCLIC MPI::DISTRIBUTE_DCK MPI:DISTRIBUTE_OFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_ORDER_C MPI::DISTRIBUTE_NONE MPI_ORDER_C MPI::ORDER_CONTRAN MPI_ORDER_FORTRAN MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_SET MPI:SEEK_SET MPI::SEEK_SET	MPI_THREAD_MULTIPLE	MPI::THREAD_MULTIPLE		
File Operation Constants, Part 1 C type: const MPI_Offset (or unnamed enum) C++ type: Cortran type: INTEGER (KIND=MPI_OFFSET_KIND) const MPI::Offset (or unnamed PI_OFFSET_KIND) MPI::DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT File Operation Constants, Part 2 C type: const int (or unnamed enum) Portran type: INTEGER const int (or unnamed enum) MPI_DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_OFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_DISTRIBUTE_OFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_ORDER_C MPI::ORDER_C MPI_ORDER_C MPI::ORDER_C MPI_ORDER_FORTRAN MPI::ORDER_FORTRAN MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_SEEK_SET MPI::SEEK_SET MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS	MPI_THREAD_SERIALIZED	MPI::THREAD_SERIALIZED		
C type: const MPI_Offset (or unnamed enum) C++ type: Cortran type: INTEGER (KIND=MPI_OFFSET_KIND) const MPI::Offset (or unnamed enum) API_DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT MPI_DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_DFLT_DARG MPI::DISTRIBUTE_CYCLIC MPI_DISTRIBUTE_DFLT_DARG MPI::DISTRIBUTE_NONE MPI_ORDER_C MPI::ORDER_C MPI_ORDER_C MPI::ORDER_C MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_CUR MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER	MPI_THREAD_SINGLE	MPI::THREAD_SINGLE		
Gentran type: INTEGER (KIND=MPI_OFFSET_KIND) const MPI::Offset (or unnan MPI::DISPLACEMENT_CURRENT MPI::DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI::DISTRIBUTE_CYCLIC MPI::DISTRIBUTE_OFLT_DARG MPI::ONDE MPI::ONDE MPI::ONDE MPI::ORDER_C MPI::ORDER_FORTRAN MPI::SEEK_CUR MPI::SEEK_SET MPI::SEEK_SET MPI::SEEK_SET <td colspan<="" th=""><th>-</th><th></th></td>	<th>-</th> <th></th>	-		
IPI_DISPLACEMENT_CURRENT MPI::DISPLACEMENT_CURRENT File Operation Constants, Part 2 File Operation Constants, Part 2 C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_DISTRIBUTE_BLOCK MPI::DISTRIBUTE_BLOCK MPI_DISTRIBUTE_OFLT_DARG MPI::DISTRIBUTE_OFLT_DARG MPI_DISTRIBUTE_NONE MPI::DISTRIBUTE_NONE MPI_ORDER_C MPI::ORDER_C MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_CUR MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI::TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER		v -		
File Operation Constants, Part 2C type: const int (or unnamed enum)C++ type:Fortran type: INTEGERconst int (or unnamed enum)MPI_DISTRIBUTE_BLOCKMPI::DISTRIBUTE_BLOCKMPI_DISTRIBUTE_CYCLICMPI::DISTRIBUTE_CYCLICMPI_DISTRIBUTE_DFLT_DARGMPI::DISTRIBUTE_DFLT_DARGMPI_DISTRIBUTE_NONEMPI::DISTRIBUTE_NONEMPI_ORDER_CMPI::ORDER_CMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_CURMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: const int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI:TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	* -			
C type: const int (or unnamed enum)C++ type: const int (or unnamed enum)Fortran type: INTEGERconst int (or unnamed enum)MPI_DISTRIBUTE_BLOCKMPI::DISTRIBUTE_BLOCKMPI_DISTRIBUTE_CYCLICMPI::DISTRIBUTE_CYCLICMPI_DISTRIBUTE_DFLT_DARGMPI::DISTRIBUTE_DFLT_DARGMPI_ORDER_CMPI::DISTRIBUTE_NONEMPI_ORDER_FORTRANMPI::ORDER_FORTRANMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_ENDMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: const int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	MPI_DISPLACEMENT_CURRENT	MPI::DISPLACEMENT_CURREN		
C type: const int (or unnamed enum)C++ type: const int (or unnamed enum)Fortran type: INTEGERconst int (or unnamed enum)MPI_DISTRIBUTE_BLOCKMPI::DISTRIBUTE_BLOCKMPI_DISTRIBUTE_CYCLICMPI::DISTRIBUTE_CYCLICMPI_DISTRIBUTE_DFLT_DARGMPI::DISTRIBUTE_DFLT_DARGMPI_ORDER_CMPI::DISTRIBUTE_NONEMPI_ORDER_FORTRANMPI::ORDER_FORTRANMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_ENDMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: const int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	File Operation Cons	stants, Part 2		
MPI_DISTRIBUTE_BLOCKMPI::DISTRIBUTE_BLOCKMPI_DISTRIBUTE_CYCLICMPI::DISTRIBUTE_CYCLICMPI_DISTRIBUTE_DFLT_DARGMPI::DISTRIBUTE_DFLT_DARGMPI_DISTRIBUTE_NONEMPI::DISTRIBUTE_NONEMPI_ORDER_CMPI::ORDER_CMPI_ORDER_FORTRANMPI::ORDER_FORTRANMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_ENDMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: const int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	C type: const int (or unnamed enum)	C++ type:		
MPI_DISTRIBUTE_CYCLICMPI::DISTRIBUTE_CYCLICMPI_DISTRIBUTE_DFLT_DARGMPI::DISTRIBUTE_DFLT_DARGMPI_DISTRIBUTE_NONEMPI::DISTRIBUTE_NONEMPI_ORDER_CMPI::ORDER_CMPI_ORDER_FORTRANMPI::ORDER_FORTRANMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_ENDMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: const int (or unnamed enum)C type: INTEGERconst int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_INTEGERMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	Fortran type: INTEGER	const int (or unnamed enum)		
MPI_DISTRIBUTE_DFLT_DARGMPI::DISTRIBUTE_DFLT_DARGMPI_DISTRIBUTE_NONEMPI::DISTRIBUTE_NONEMPI_ORDER_CMPI::ORDER_CMPI_ORDER_FORTRANMPI::ORDER_FORTRANMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_ENDMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: INTEGERconst int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_INTEGER		MPI::DISTRIBUTE_BLOCK		
MPI_DISTRIBUTE_NONEMPI::DISTRIBUTE_NONEMPI_ORDER_CMPI::ORDER_CMPI_ORDER_FORTRANMPI::ORDER_FORTRANMPI_SEEK_CURMPI::SEEK_CURMPI_SEEK_ENDMPI::SEEK_ENDMPI_SEEK_SETMPI::SEEK_SETF90 Datatype Matching ConstantsC type: const int (or unnamed enum)C type: const int (or unnamed enum)C++ type:Fortran type: INTEGERconst int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	MPI_DISTRIBUTE_CYCLIC	MPI::DISTRIBUTE_CYCLIC		
MPI_ORDER_C MPI::ORDER_C MPI_ORDER_FORTRAN MPI::ORDER_FORTRAN MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER	MPI_DISTRIBUTE_DFLT_DARG	MPI::DISTRIBUTE_DFLT_DARG		
MPI_ORDER_FORTRAN MPI::ORDER_FORTRAN MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER	MPI_DISTRIBUTE_NONE	MPI::DISTRIBUTE_NONE		
MPI_ORDER_FORTRAN MPI::ORDER_FORTRAN MPI_SEEK_CUR MPI::SEEK_CUR MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER				
MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER	MPI_ORDER_FORTRAN	MPI::ORDER_FORTRAN		
MPI_SEEK_END MPI::SEEK_END MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER	MPI_SEEK_CUR	MPI::SEEK_CUR		
MPI_SEEK_SET MPI::SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) C++ type: Fortran type: INTEGER const int (or unnamed enum) MPI_TYPECLASS_COMPLEX MPI::TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER				
C type: const int (or unnamed enum)C++ type:Fortran type: INTEGERconst int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER	MPI_SEEK_SET	MPI::SEEK_SET		
C type: const int (or unnamed enum)C++ type:Fortran type: INTEGERconst int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER				
Fortran type: INTEGERconst int (or unnamed enum)MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER		ing Constants		
MPI_TYPECLASS_COMPLEXMPI::TYPECLASS_COMPLEXMPI_TYPECLASS_INTEGERMPI::TYPECLASS_INTEGER		C++ type:		
MPI_TYPECLASS_INTEGER MPI::TYPECLASS_INTEGER	V I			
MPI_TYPECLASS_REAL MPI::TYPECLASS_REAL				
	MPI_TYPECLASS_REAL	MPI::TYPECLASS_REAL		

C/Fortran name	Empty or Ignored Input C++ name	2
		2
C type / Fortran type	C++ type	4
MPI_ARGVS_NULL	MPI::ARGVS_NULL	-
char*** / 2-dim. array of CHARACTER*(5
MPI_ARGV_NULL	MPI::ARGV_NULL	6
<pre>char** / array of CHARACTER*(*)</pre>	const char **	7
MPI_ERRCODES_IGNORE	Not defined for C++	8
<pre>int* / INTEGER array</pre>		9
MPI_STATUSES_IGNORE	Not defined for C++	10
MPI_Status* / INTEGER, DIMENSION(M)	•	11
[ticket231-C.] or TYPE(MPI_Status), D	IMENSION(*)	12
MPI_STATUS_IGNORE	Not defined for C++	13
MPI_Status* / INTEGER, DIMENSION(M)	PI_STATUS_SIZE)	14
[ticket231-C.] or TYPE(MPI_Status)		15
MPI_UNWEIGHTED	Not defined for C++	16
[ticket0.172.] int* / INTEGER array		17
		18
		10
		19
C Constants Specifying I	gnored Input (no C++ or Fortran)	
	gnored Input (no C++ or Fortran) [ticket243-O.]equivalent to Fortran	19
C type: MPI_Fint*	[ticket243-O.]equivalent to Fortran	19 20 21
C type: MPI_Fint* MPI_F_STATUSES_IGNORE	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi /	19 20 21 mpif.h
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp	19 20 21 mpif.h
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status*	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran	19 20 21 mpif.h oif.h 24
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f	19 20 21 mpif.h oif.h 24 08 25
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran	19 20 21 mpif.h 08 25 08 25
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f	$ \begin{array}{r} 19 \\ 20 \\ 21 \\ mpif.h \\ pif.h \\ 24 \\ 08 \\ 25 \\ 26 \\ 26 \end{array} $
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE ticket243-O.]MPI_F08_STATUS_IGNORE	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f [ticket243-O.]MPI_STATUS_IGNORE in mpi_f08	19 20 21 mpif.h oif.h 24 08 25 26 27
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE ticket243-O.]MPI_F08_STATUS_IGNORE C and C++ preprocessor Cor	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f [ticket243-O.]MPI_STATUS_IGNORE in mpi_f08	$ \begin{array}{r} 19 \\ 20 \\ 21 \\ mpif.h \\ pif.h \\ 24 \\ 24 \\ 25 \\ 26 \\ 27 \\ 28 \\ \end{array} $
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE ticket243-O.]MPI_F08_STATUS_IGNORE $\frac{C \text{ and } C++ \text{ preprocessor Corr}}{C/C++ \text{ type: const int (or unname)}}$	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f [ticket243-O.]MPI_STATUS_IGNORE in mpi_f08	19 20 21 mpif.h 24 08 25 26 27 28 29
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE ticket243-O.]MPI_F08_STATUS_IGNORE $\frac{C \text{ and } C++ \text{ preprocessor Cond}}{C/C++ \text{ type: const int (or unname)}}$	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f [ticket243-O.]MPI_STATUS_IGNORE in mpi_f08	19 20 21 mpif.h oif.h 24 08 25 26 27 28 29 30
C type: MPI_Fint* MPI_F_STATUSES_IGNORE MPI_F_STATUS_IGNORE ticket243-O.]C type: MPI_F08_status* ticket243-O.]MPI_F08_STATUSES_IGNORE ticket243-O.]MPI_F08_STATUS_IGNORE $\frac{C \text{ and } C++ \text{ preprocessor Corr}}{C/C++ \text{ type: const int (or unname)}}$	[ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi / [ticket243-O.]MPI_STATUS_IGNORE in mpi / mp [ticket243-O.]equivalent to Fortran [ticket243-O.]MPI_STATUSES_IGNORE in mpi_f [ticket243-O.]MPI_STATUS_IGNORE in mpi_f08	$ \begin{array}{c} 19\\ 20\\ \hline 21\\ mpif.h\\ 24\\ \hline 24\\ \hline 25\\ 26\\ \hline 27\\ 28\\ 29\\ 30\\ 31\\ \hline \end{array} $

Null handles used in the MPI tool information interface
MPI_T_ENUM_NULL
MPI_T_CVAR_HANDLE_NULL
MPI_T_PVAR_HANDLE_NULL
MPI_T_PVAR_SESSION_NULL

1	Verbesity Levels in the MDI tool information interface
2	Verbosity Levels in the MPI tool information interface
3	
4	MPI_T_VERBOSITY_USER_ALL
5	MPI_T_VERBOSITY_TUNER_BASIC
6	MPI_T_VERBOSITY_TUNER_DETAIL
7	MPI_T_VERBOSITY_TUNER_ALL
8	MPI_T_VERBOSITY_MPIDEV_BASIC
9	MPI_T_VERBOSITY_MPIDEV_DETAIL
0	MPI_T_VERBOSITY_MPIDEV_ALL
1	
3	Constants to identify associations of variables
4	in the MPI tool information interface
5	MPI_T_BIND_NO_OBJECT
6	MPI_T_BIND_MPI_COMM
7	MPI_T_BIND_MPI_DATATYPE
8	MPI_T_BIND_MPI_ERRHANDLER
9	MPI_T_BIND_MPI_FILE
0	
1	MPI_T_BIND_MPI_GROUP
	MPI_T_BIND_MPI_OP
2	MPI_T_BIND_MPI_REQUEST
3	MPI_T_BIND_MPI_WIN
4	MPI_T_BIND_MPI_MESSAGE
5	MPI_T_BIND_MPI_INFO
6	
7	
8	Constants describing the scope of a control variable
9	in the MPI tool information interface
0	MPI_T_SCOPE_READONLY
1	MPI_T_SCOPE_LOCAL
2	MPI_T_SCOPE_GROUP
3	MPI_T_SCOPE_GROUP_EQ
4	MPI_T_SCOPE_GLOBAL
5	MPI_T_SCOPE_GLOBAL_EQ
3	
7	
5	
)	Additional constants used
)	by the MPI tool information interface
	MPI_T_PVAR_ALL_HANDLES
2	
2	
3	
4	
5	
6	
17	
18	

	Performance variables classes used by the	1
	MPI tool information interface	2
	MPI_T_PVAR_CLASS_STATE	3
	MPI_T_PVAR_CLASS_LEVEL	4
	MPI_T_PVAR_CLASS_SIZE	5
	MPI_T_PVAR_CLASS_PERCENTAGE	6
	MPI_T_PVAR_CLASS_HIGHWATERMARK	7
	MPI_T_PVAR_CLASS_LOWWATERMARK	8
	MPI_T_PVAR_CLASS_COUNTER	9
	MPI_T_PVAR_CLASS_AGGREGATE	10
	MPI_T_PVAR_CLASS_TIMER	11
	MPI_T_PVAR_CLASS_GENERIC	12
		13
		14
A.1.2 Types		15
		16
The following are de	efined C type definitions, included in the file mpi.h.	17
/* C opaque types	s */	18
MPI_Aint		19
MPI_Count		20 ticket 265.
MPI_Fint		21
MPI_Offset		22
MPI_Status		23
MPI_F08_status		²⁴ ticket243-O.
		25
/* C handles to a	assorted structures */	26
MPI_Comm		27
MPI_Datatype		28
MPI_Errhandler		29
MPI_File		30
MPI_Group		31
MPI_Info		32
MPI_Message		33 ticket274.
MPI_Op		34
MPI_Request		35
MPI_Win		36
		37 ticket 266.
/* Types for the	MPI_T interface */	38
MPI_T_enum		39
MPI_T_cvar_handle		40
MPI_T_pvar_handle		41
MPI_T_pvar_session		42
	<u>20</u>	43
		43
// C++ 0000000 +	and (all within the MPI namespace)	44
MPI::Aint	pes (all within the MPI namespace)	45
		46
MPI::Offset		
MPI::Status		48

```
1
            \mathbf{2}
                  // C++ handles to assorted structures (classes,
            3
                  // all within the MPI namespace)
            4
                  MPI::Comm
            5
                  MPI::Intracomm
            6
                  MPI::Graphcomm
            7
                  MPI::Distgraphcomm
            8
                  MPI::Cartcomm
            9
                  MPI::Intercomm
            10
                  MPI::Datatype
            11
                  MPI::Errhandler
            12
                  MPI::Exception
            13
                  MPI::File
            14
                  MPI::Group
            15
                  MPI::Info
            16
                  MPI::Op
            17
                  MPI::Request
            18
                  MPI::Prequest
            19
                  MPI::Grequest
            20
                  MPI::Win
ticket243-O.<sup>21</sup>
            22
                      The following are defined Fortran type definitions, included in the mpi_f08 and mpi
            23
                  module.
            ^{24}
                  ! Fortran opaque types in the mpi_f08 and mpi module
            25
                  TYPE(MPI_Status)
            26
            27
ticket231-C.
                     Fortran handles in the mpi_f08 and mpi module
                  1
            28
                  TYPE(MPI_Comm)
            29
                  TYPE(MPI_Datatype)
            30
                  TYPE(MPI_Errhandler)
            31
                 TYPE(MPI_File)
            32
                  TYPE(MPI_Group)
            33
                 TYPE(MPI_Info)
            34
                  TYPE(MPI_Op)
            35
                 TYPE(MPI_Request)
            36
                 TYPE(MPI_Win)
            37
            38
                  A.1.3 Prototype [d] Definitions
    ticket0. 39
ticket230-B. _{40}
                  C Bindings
            41
                  The following are defined C typedefs for user-defined functions, also included in the file
            42
                  mpi.h.
            43
            44
                  /* prototypes for user-defined functions */
            45
                  typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
            46
                                  MPI_Datatype *datatype);
            47
            48
```

<pre>typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm,</pre>	1
<pre>int comm_keyval, void *extra_state, void *attribute_val_in,</pre>	2
<pre>void *attribute_val_out, int*flag);</pre>	3
<pre>typedef int MPI_Comm_delete_attr_function(MPI_Comm comm,</pre>	4
<pre>int comm_keyval, void *attribute_val, void *extra_state);</pre>	5
	6
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,	7
void *extra_state, void *attribute_val_in,	8
<pre>void *attribute_val_out, int *flag);</pre>	9
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,	10
void *attribute_val, void *extra_state);	11
	12
<pre>typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,</pre>	13
int type_keyval, void *extra_state,	14
<pre>void *attribute_val_in, void *attribute_val_out, int *flag);</pre>	15
typedef int MPI_Type_delete_attr_function(MPI_Datatype [ticket252-W.]datatype,	16
int type_keyval, void *attribute_val, void *extra_state);	17
	18
<pre>typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *,);</pre>	19
typedef void MPI_Win_errhandler_function(MPI_Win *, int *,);	20
typedef void MPI_File_errhandler_function(MPI_File *, int *,);	21
	22
<pre>typedef int MPI_Grequest_query_function(void *extra_state,</pre>	23
MPI_Status *status);	24
<pre>typedef int MPI_Grequest_free_function(void *extra_state);</pre>	25
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);	26
	27
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,	28
MPI_Aint *file_extent, void *extra_state);	29
typedef int MPI_Datarep_conversion_function(void *userbuf,	30
MPI_Datatype datatype, int count, void *filebuf,	31
MPI_Offset position, void *extra_state);	32
	³³ ticket230-B.
	34
Fortran 2008 Bindings with the mpi_f08 Module	35
	$_{36}$ ticket230-B.
With the Fortran mpi_f08 module, the callback prototypes are:	37
The user-function argument to MPI_Op_create should be declared according to:	³⁸ ticket-248T.
ABSTRACT INTERFACE	39
SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)	40
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	41
TYPE(C_PTR), VALUE :: invec, inoutvec	42
INTEGER :: len	43
TYPE(MPI_Datatype) :: datatype	44
	$_{45}$ ticket 230-B.
The copy and delete function arguments to MPI_Comm_create_keyval should be de-	46
clared according to:	$_{\rm 47}$ ticket-248T.
ABSTRACT INTERFACE	48

	1	SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
	2	attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
	3	TYPE(MPI_Comm) :: oldcomm
	4	INTEGER :: comm_keyval, ierror
	5	INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
	6	attribute_val_out
	7	LOGICAL :: flag
ticket-248T.	• 8	
	9	ABSTRACT INTERFACE
	10	SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
	11	attribute_val, extra_state, ierror) BIND(C)
	12	TYPE(MPI_Comm) :: comm
	13	INTEGER :: comm_keyval, ierror
ticket230-B.	14	INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
tionet200 D	15	The copy and delete function arguments to MPI_Win_create_keyval should be declared
ticket-248T.	. 16	according to:
	17	ABSTRACT INTERFACE
	18	SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
	19	attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
	20	TYPE(MPI_Win) :: oldwin
	21	INTEGER :: win_keyval, ierror
	22	<pre>INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,</pre>
	23	attribute_val_out
1.1.4.04000	24	LOGICAL :: flag
ticket-248T.		ABSTRACT INTERFACE
	26	SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
	27	extra_state, ierror) BIND(C)
	28	
		IYPEUMPI WIND II WIN
	29	TYPE(MPI_Win) :: win INTEGER :: win kevval, ierror
	30	INTEGER :: win_keyval, ierror
ticket230-B.	30 31	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state</pre>
	30 31 32	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared</pre>
ticket230-B. ticket-248T.	30 31 32 33	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to:</pre>
	30 31 32 33 34	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE</pre>
	30 31 32 33	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,</pre>
	30 31 32 33 33 34 35	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C)</pre>
	30 31 32 33 34 35 36	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype</pre>
	30 31 32 33 34 35 36 37	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror</pre>
	30 31 32 33 34 35 36 37 38	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,</pre>
	30 31 32 33 34 35 36 37 38 39	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out</pre>
	30 31 32 33 34 35 36 37 38 39 40 41	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out LOGICAL :: flag</pre>
ticket-248T.	30 31 32 33 34 35 36 37 38 39 40 41	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out LOGICAL :: flag</pre>
ticket-248T.	30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out LOGICAL :: flag</pre>
ticket-248T.	30 31 32 33 34 35 36 37 38 39 40 41 . 42 43	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out LOGICAL :: flag</pre>
ticket-248T.	30 31 32 33 34 35 36 37 38 39 40 41 41 42 43	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out LOGICAL :: flag</pre> ABSTRACT INTERFACE SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval, attribute_val, extra_state, ierror) BIND(C) TYPE(MPI_Datatype) :: datatype
ticket-248T.	30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) TYPE(MPI_Datatype) :: oldtype INTEGER :: type_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in, attribute_val_out LOGICAL :: flag</pre> ABSTRACT INTERFACE SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval, attribute_val, extra_state, ierror) BIND(C)

	$^{1}_{2}$ ticket230-B.
The handler-function argument to MPI_Comm_create_errhandler should be declared like this:	² ticket-248T.
ABSTRACT INTERFACE	³ ticket-2481.
SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)	5
TYPE(MPI_Comm) :: comm	6
INTEGER :: error_code	7
The handler-function argument to MPI_Win_create_errhandler should be declared like	$_{8}$ ticket 230-B.
this:	$\frac{9}{2}$ ticket-248T.
ABSTRACT INTERFACE	10
SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)	11
TYPE(MPI_Win) :: win	12
INTEGER :: error_code	¹⁴ ticket230-B.
The handler-function argument to MPI_File_create_errhandler should be declared like	15
this:	16 ticket-248T.
ABSTRACT INTERFACE	17
SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)	18
TYPE(MPI_File) :: file	19
INTEGER :: error_code	$^{20}_{21}$ ticket230-B.
The query, free, and cancel function arguments to MPI_Grequest_start should be de-	22
clared according to:	$^{22}_{23}$ ticket-248T.
ABSTRACT INTERFACE	24
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)	25
BIND(C)	26
TYPE(MPI_Status) :: status	27
INTEGER :: ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	28
INTEGER(KIND-MIT_ADDREDS_KIND) extra_State	29 ticket-248T.
ABSTRACT INTERFACE	30
SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)	31 32
INTEGER :: ierror	33
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	$_{34}$ ticket-248T.
ABSTRACT INTERFACE	35
SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)	36
BIND(C)	37
INTEGER :: ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	38
LOGICAL :: complete	39
	40 ticket230-B.
The extend and conversion function arguments to MPI_Register_datarep should be de-	41
clared according to:	⁴² ticket229.1. ⁴³ ticket-248T.
ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,	⁴⁰ ticket-2481.
ierror) BIND(C)	45
TYPE(MPI_Datatype) :: datatype	46
INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state	47
· –	48

ticket-248T.	
-	
3	SUBBOUTINE MPI Dataren conversion function(userbuf datatyne count
Ę	filebuf position extra state ierror) BIND(C)
e e	USE INTRINSIC ··· ISO C BINDING ONLY · C PTR
-	TYPE(C_PTR), VALUE :: userbuf, filebuf
٤	
ç	
1	
ticket 230-B. $^{\rm 1}$	1 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
1 ticket230-B. 1	Tortrait Dinulings with hipfi.if of the hipfi Module
11CKet230-D. 1	[For Fortran] With the Fortran mai module or maif he have are examples of how each
1	of the user defined subroutings should be dealered
	⁶ The user-function argument to MPI_OP_CREATE should be declared like this:
1	⁷ SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, [ticket252-W.]DATATYPE)
1	<pre>8 <type> INVEC(LEN), INOUTVEC(LEN)</type></pre>
1	
2	⁰ The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
2	declared like these:
	2
2	³ SUBROUTINE COMM_COPY_ATTR_[ticket250-V.][FN]FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
2	ATTRIBUTE VAL IN. ATTRIBUTE VAL OUT. FLAG. IERROR)
2	6 INTEGER OLDCOMM, COMM_KEYVAL, IERROR
2	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
2	8 ATTRIBUTE_VAL_OUT
2	9 LOGICAL FLAG
3	SUBROUTINE COMM_DELETE_ATTR_[ticket250-V.][FN]FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
3	EXTRA STATE, IERROR)
3	INTEGER COMM, COMM_KEYVAL, IERROR
	$\frac{1}{4}$ INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
3	
3	[°] The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be ⁶ declared like these:
3	7
3	⁸ SUBROUTINE WIN_COPY_ATTR_[ticket250-V.][FN]FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
3	⁹ ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
	^o INTEGER OLDWIN, WIN_KEYVAL, IERROR
4	INTEGER(KIND-MIT_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
	² ATTRIBUTE_VAL_OUT ³ LOCICAL FLAC
	³ LOGICAL FLAG 4
	⁵ SUBROUTINE WIN_DELETE_ATTR_[ticket250-V.][FN]FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
	⁶ EXTRA_STATE, IERROR)
4	
4	⁸ INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

The copy and delete function arguments to $MPI_TYPE_CREATE_KEYVAL$ should be declared like these:	1 2
SUBROUTINE TYPE_COPY_ATTR_[ticket250-V.][FN]FUNCTION(OLDTYPE, TYPE_KEYVAL, EXT ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	³ RA_STATE,
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	6
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,	7
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT	8
LOGICAL FLAG	9
	10
SUBROUTINE TYPE_DELETE_ATTR_[ticket250-V.][FN]FUNCTION([ticket252-W.]DATATYPE, EXTRA_STATE, IERROR)	TYPE_KEYVAL ,
INTEGER [ticket252-W.]DATATYPE, TYPE_KEYVAL, IERROR	13
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	14
The handler function appropriate MDL COMM CREATE EDDHANDLED should be de	15
The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:	16
clared like this.	17 18
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)	19
INTEGER COMM, ERROR_CODE	20
	21
The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-	22
clared like this:	23
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)	24
INTEGER WIN, ERROR_CODE	25
INTEGER WIN, ERROR_ODDE	26
The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-	27
clared like this:	28
	29 30
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)	31
INTEGER FILE, ERROR_CODE	32
The query, free, and cancel function arguments to MPI_GREQUEST_START should be	33
declared like these:	34
	35
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)	36
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	37
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	38
	39
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)	40
INTEGER IERROR	41 42
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	42
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)	44
INTEGER IERROR	45
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	46
LOGICAL COMPLETE	47
	48

1 The extend and conversion function arguments to MPI_REGISTER_DATAREP should $\mathbf{2}$ be declared like these: 3 SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) 4 INTEGER DATATYPE, IERROR 5INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE 6 7 SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF, 8 POSITION, EXTRA_STATE, IERROR) 9 <TYPE> USERBUF(*), FILEBUF(*) 10 INTEGER COUNT, DATATYPE, IERROR 11 INTEGER(KIND=MPI_OFFSET_KIND) POSITION 12 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 13 ticket230-B. 14 15 C++ Bindings (deprecated) 16 ticket185. 17 The following are deprecated defined C++ typedefs, also included in the file mpi.h. 18 namespace MPI { 19 typedef void User_function(const void* invec, void *inoutvec, 20int len, const Datatype& datatype); 2122typedef int Comm::Copy_attr_function(const Comm& oldcomm, 23int comm_keyval, void* extra_state, void* attribute_val_in, 24void* attribute_val_out, bool& flag); 25typedef int Comm::Delete_attr_function(Comm& comm, int 26comm_keyval, void* attribute_val, void* extra_state); 27typedef int Win::Copy_attr_function(const Win& oldwin, 2829 int win_keyval, void* extra_state, void* attribute_val_in, 30 void* attribute_val_out, bool& flag); 31typedef int Win::Delete_attr_function(Win& win, int 32 win_keyval, void* attribute_val, void* extra_state); 33 34 typedef int Datatype::Copy_attr_function(const Datatype& oldtype, 35int type_keyval, void* extra_state, 36 const void* attribute_val_in, void* attribute_val_out, 37 bool& flag); 38 typedef int Datatype::Delete_attr_function(Datatype& [ticket3.].0}{252-W}{data}type, 39 int type_keyval, void* attribute_val, void* extra_state); 4041 typedef void Comm::Errhandler_function(Comm &, int *, ...); 42typedef void Win::Errhandler_function(Win &, int *, ...); 43 typedef void File::Errhandler_function(File &, int *, ...); 44 45typedef int Grequest::Query_function(void* extra_state, Status& status); 46typedef int Grequest::Free_function(void* extra_state); 47 typedef int Grequest::Cancel_function(void* extra_state, bool complete); 48

```
1
  typedef void Datarep_extent_function(const Datatype& datatype,
                                                                                          \mathbf{2}
                 Aint& file_extent, void* extra_state);
                                                                                          3
  typedef void Datarep_conversion_function(void* userbuf,
                                                                                          4
                 Datatype& datatype, int count, void* filebuf,
                 Offset position, void* extra_state);
                                                                                          5
}
                                                                                          6
                                                                                          7
                                                                                          <sup>8</sup> ticket0.
       Deprecated [p]Prototype [d]Definitions
A.1.4
                                                                                          <sup>9</sup> ticket0.
The following are defined C typedefs for deprecated user-defined functions, also included in
                                                                                          10
the file mpi.h.
                                                                                          11
                                                                                          12
/* prototypes for user-defined functions */
                                                                                          13
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                                                                                          14
                void *extra_state, void *attribute_val_in,
                                                                                          15
               void *attribute_val_out, int *flag);
                                                                                          16
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
                                                                                          17
                void *attribute_val, void *extra_state);
                                                                                          18
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);
                                                                                          19
                                                                                          20
    The following are deprecated Fortran user-defined callback subroutine prototypes. The
                                                                                          21
deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-
                                                                                          22
clared like these:
                                                                                          23
                                                                                          24
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
                                                                                          25
                 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
                                                                                          26
   INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                          27
          ATTRIBUTE_VAL_OUT, IERR
                                                                                          28
   LOGICAL FLAG
                                                                                          29
                                                                                          30
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
                                                                                          31
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
                                                                                          32
                                                                                          33
    The deprecated handler-function for error handlers should be declared like this:
                                                                                          34
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE)
                                                                                          35
   INTEGER COMM, ERROR_CODE
                                                                                          36
                                                                                          37
                                                                                          38
A.1.5 Info Keys
                                                                                          39
access_style
                                                                                          40
appnum
                                                                                          41
arch
                                                                                          42
cb_block_size
                                                                                          43
cb_buffer_size
                                                                                          44
cb_nodes
                                                                                          45
chunked_item
                                                                                          46
chunked_size
                                                                                          47
chunked
                                                                                          48
```

- ² file_perm
- ³ filename
- ⁴ file
- ⁵ host
- ⁶ io_node_list
- ⁷ ip_address
- ⁸ ip_port
- ⁹ nb_proc
- ¹⁰ no_locks
- ¹¹ num_io_nodes
- ¹² path
- ¹³ soft
- ¹⁴ striping_factor
- ¹⁵ striping_unit
- ¹⁶ wdir
- 17
- 18

¹⁹ A.1.6 Info Values

- 20 21 false
- 21 random
- read_mostly
- read_once
- 25 reverse_sequential
- 26 sequential
- 27 true
- 28 write_mostly
- 29 write_once
- 30
- 31
- 32
- 33 34
- 35
- 36
- 37
- 38 39
- 40
- 41
- 42 43
- 44
- 45
- 46
- 47
- 48

A.2 C Bindings	1 2
A.2.1 Point-to-Point Communication C Bindings	3
<pre>int MPI_Bsend_init(const void* buf, int count, MPI_Datatype datatype,</pre>	$\frac{4}{5}$ ticket140.
<pre>int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	7 ticket140. 8
<pre>int MPI_Buffer_attach(void* buffer, int size)</pre>	9 10
<pre>int MPI_Buffer_detach(void* buffer_addr, int* size)</pre>	11
<pre>int MPI_Cancel(MPI_Request *request)</pre>	12 13
<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>	¹⁴ ticket140.
<pre>int MPI_Ibsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	$^{10}_{17}$ ticket140.
int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status)	19 20 21
<pre>int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)</pre>	21 22 23
<pre>int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)</pre>	24 25 26
<pre>int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,</pre>	27 28
<pre>int MPI_Irsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	²⁹ ticket140. ³⁰
<pre>int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	32 ticket140.
<pre>int MPI_Issend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	$^{34}_{35}$ ticket140.
<pre>int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)</pre>	37 38
<pre>int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)</pre>	39 40 41
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	42
<pre>int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,</pre>	43 44 45
<pre>int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,</pre>	46 47 48

	738 ANNEX A. LANGUAGE BINDINGS SUMMARY
1	<pre>int MPI_Request_free(MPI_Request *request)</pre>
2 3 4	<pre>int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)</pre>
ticket140. 5 6 7	<pre>int MPI_Rsend_init(const void* buf, int count, MPI_Datatype datatype,</pre>
ticket140. $\frac{1}{8}$	<pre>int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
ticket140. $^{10}_{11}_{12}$	<pre>int MPI_Send_init(const void* buf, int count, MPI_Datatype datatype,</pre>
ticket140. 13	<pre>int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
ticket 140. $\frac{15}{16}$	<pre>int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>
20 21 22	<pre>int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,</pre>
ticket140. $^{^{23}}_{_{24}}$	<pre>int MPI_Ssend_init(const void* buf, int count, MPI_Datatype datatype,</pre>
ticket140. 26 27	<pre>int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
${{ m ticket125.}}^{28}_{29}$	<pre>int MPI_Startall(int count, MPI_Request [*]array_of_requests[])</pre>
30	<pre>int MPI_Start(MPI_Request *request)</pre>
ticket125. ³¹ ticket125. ³² ticket125. ³³	<pre>int MPI_Testall(int count, MPI_Request [*]array_of_requests[], int *flag, MPI_Status [*]array_of_statuses[])</pre>
ticket 125. ³⁴ ticket 125. ³⁵	<pre>int MPI_Testany(int count, MPI_Request [*]array_of_requests[], int *index,</pre>
$ticket 125. {}^{36}$ $ticket 140. {}^{37}$	<pre>int MPI_Test_cancelled(const MPI_Status *status, int *flag)</pre>
38	<pre>int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)</pre>
39 ticket125. 40 ticket125. 41 ticket125. 42 ticket125. 42 ticket125.	<pre>int MPI_Testsome(int incount, MPI_Request [*]array_of_requests[],</pre>
ticket125. ⁴³ ticket125. ⁴⁴	<pre>int MPI_Waitall(int count, MPI_Request [*]array_of_requests[],</pre>
ticket125. ⁴⁵ ticket125. ⁴⁶ ticket125. ⁴⁷ ticket125. ⁴⁸ ticket125.	int MPI_Waitany(int count, MPI_Request [*]array_of_requests[], int *index, MPI_Status *status)
ticket125.	Unofficial Draft for Comment Only

int MPI_Wait(MPI_Request *request, MPI_Status *status)	1
<pre>int MPI_Waitsome(int incount, MPI_Request [*]array_of_requests[],</pre>	2_3 ticket125. 4_4 ticket125. 5_5 ticket125. 6_6 ticket125.
A.2.2 Datatypes C Bindings	$\frac{7}{8}$ ticket125.
<pre>int MPI_Get_address(const void *location, MPI_Aint *address)</pre>	9 ticket140.
<pre>int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>	$^{10}_{11}$ ticket140.
<pre>int MPI_Get_elements_x(MPI_Status *status, MPI_Datatype datatype, MPI_Count *count)</pre>	13 14
<pre>int MPI_Pack_external(const char *datarep, const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint *position)</pre>	$^{15}_{16}$ ticket140. ticket140.
int MPI_Pack_external_size(<mark>const</mark> char *datarep, int incount, MPI_Datatype datatype, MPI_Aint *size)	19 ticket140.
<pre>int MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype,</pre>	$^{21}_{22}$ ticket 140.
<pre>int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,</pre>	24 25
<pre>int MPI_Type_commit(MPI_Datatype *datatype)</pre>	26 27
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)	28 29
<pre>int MPI_Type_create_darray(int size, int rank, int ndims, const</pre>	${^{30}_{31}}$ ticket140. ${^{32}_{32}}$ ticket140. ${^{33}_{33}}$ ticket140. ${^{34}}$ ticket140.
<pre>int MPI_Type_create_hindexed_block(int count, int blocklength,</pre>	35 36 37
<pre>int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],</pre>	$^{38}_{39}$ ticket140. $_{40}$ ticket140. 41
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)	42 43 44
<pre>int MPI_Type_create_indexed_block(int count, int blocklength, const</pre>	44 45 ticket140. 46 47 48

	1 2	<pre>int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint</pre>
ticket140. ticket140. ticket140.	5	<pre>int MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>
ticket140. ticket140. ticket140.	8 9	<pre>int MPI_Type_create_subarray(int ndims, const int array_of_sizes[], const</pre>
ticket 252-W.	10 11	<pre>int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>
	12	<pre>int MPI_Type_free(MPI_Datatype *datatype)</pre>
	13 14 15 16 17	<pre>int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,</pre>
	18 19 20	<pre>int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>
	21 22	<pre>int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,</pre>
	23 24 25	<pre>int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,</pre>
ticket140. ticket140.		<pre>int MPI_Type_indexed(int count, const int *array_of_blocklengths, const</pre>
	29 30	<pre>int MPI_Type_size(MPI_Datatype datatype, int *size)</pre>
	31 32	<pre>int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>
ticket140. ticket140.	33 34 35 36	<pre>int MPI_Unpack_external(const char *datarep, const void *inbuf,</pre>
ticket140.	37 38 39	<pre>int MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf,</pre>
	40 41	A.2.3 Collective Communication C Bindings
ticket140.	43 44	<pre>int MPI_Allgather(const void* sendbuf, int sendcount,</pre>
ticket140. ticket140. ticket140.	47	<pre>int MPI_Allgatherv(const void* sendbuf, int sendcount,</pre>
4		

int	<pre>MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>	1 ticket140.
int	<pre>MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>	$\frac{3}{4}$ ticket140.
int	<pre>MPI_Alltoallv(const void* sendbuf, const int sendcounts[], const</pre>	 ⁷ ticket140. ⁸ ticket140. ⁹ ticket140. ¹⁰ ticket140. ¹¹ ticket140.
int	<pre>MPI_Alltoallw(const void* sendbuf, const int sendcounts[], const</pre>	¹² ticket140. ¹³ ticket140. ¹⁴ ticket140. ¹⁵ ticket140. ¹⁶ ticket140.
int	MPI_Barrier(MPI_Comm comm)	10 ticket 140.
int	<pre>MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)</pre>	$^{17}_{18}$ ticket140.
int	<pre>MPI_Exscan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>	$^{20}_{21}$ ticket 140.
int	<pre>MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	23 ticket140. 24 25
int	<pre>MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	 ²⁶ ticket140. ²⁷ ticket140. ²⁸ ticket140.
int	<pre>MPI_Iallgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	30 ticket140. 31 32
int	<pre>MPI_Iallgatherv(const void* sendbuf, int sendcount,</pre>	 ³³ ticket140. ³⁴ ticket140. ³⁵ ticket140. ³⁶ 37
int	<pre>MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,</pre>	38 ticket140. 39 40
int	<pre>MPI_Ialltoall(const void* sendbuf, int sendcount,</pre>	$^{41}_{42}$ ticket140.
int	<pre>MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	 ⁴⁵ ticket140. ⁴⁶ ticket140. ⁴⁷ ticket140. ⁴⁸ ticket140. ticket140.

ticket140. ¹ ticket140. ² ticket140. ³ ticket140. ⁴ ticket140. ₅	<pre>int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[], void* recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)</pre>
ticket 140. $_6$	<pre>int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)</pre>
${{ m ticket140.}}_{8}$	<pre>int MPI_Ibcast(const void* buffer, int count, MPI_Datatype datatype,</pre>
9 ticket140. ₁₀ 11 12	<pre>int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,</pre>
ticket140. ¹³ 14 15 16	<pre>int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
ticket140. ₁₇ ticket140. ₁₈ ticket140. ₁₉ 20	<pre>int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
ticket140. ²¹ 22 23 24	<pre>int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Request *request)</pre>
ticket140. 25 26 27	<pre>int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>
$ticket 140. {}^{28}$ $ticket 140. {}^{29}$ $_{30}$	<pre>int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>
ticket140. 32 33 34	int MPI_Iscan(<mark>const</mark> void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
$ticket 140. \frac{35}{36}$	<pre>int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
ticket140. 39 ticket140. 40 ticket140. 41 42	<pre>int MPI_Iscatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
43	<pre>int MPI_Op_commutative(MPI_Op op, int *commute)</pre>
$^{44}_{ m ticket 252-W.}$ $^{45}_{ m 46}$	<pre>int MPI_Op_create(MPI_User_function* [function]user_fn, int commute,</pre>
47 48	<pre>int MPI_Op_free(MPI_Op *op)</pre>

<pre>int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,</pre>	$^{1}_{2}$ ticket140.
<pre>int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,</pre>	3_4 ticket140.
<pre>int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>	⁶ ticket140.
<pre>int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const</pre>	⁹ ₁₀ ticket140. ₁₁ ticket140. ¹²
<pre>int MPI_Scan(const void* sendbuf, void* recvbuf, int count,</pre>	$^{13}_{14}$ ticket 140.
<pre>int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	¹⁶ ticket140. ¹⁷
<pre>int MPI_Scatterv(const void* sendbuf, const int sendcounts[], const</pre>	 ¹⁹ ticket140. ²⁰ ticket140. ²¹ ticket140. ²² ²³
A.2.4 Groups, Contexts, Communicators, and Caching C Bindings	24
<pre>int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)</pre>	25 26
<pre>int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,</pre>	27 28 29
<pre>int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)</pre>	30 31
<pre>int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)</pre>	32
<pre>int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,</pre>	33 34 35
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)	36
<pre>int MPI_Comm_free_keyval(int *comm_keyval)</pre>	37 38
<pre>int MPI_Comm_free(MPI_Comm *comm)</pre>	39
<pre>int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,</pre>	40
int *flag)	41 42
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)	43
-	44
<pre>int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)</pre>	45 46
<pre>int MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,</pre>	40
<pre>void *extra_state, void *attribute_val_in,</pre>	48

ANNEX A. LANGUAGE BINDINGS SUMMARY

1	<pre>void *attribute_val_out, int *flag)</pre>
2 3 4	<pre>int MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void *attribute_val, void *extra_state)</pre>
5	<pre>int MPI_Comm_rank(MPI_Comm comm, int *rank)</pre>
6 7	<pre>int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)</pre>
8 9	<pre>int MPI_Comm_remote_size(MPI_Comm comm, int *size)</pre>
9 10	<pre>int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)</pre>
ticket 140. $^{11}_{12}$	<pre>int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)</pre>
12	<pre>int MPI_Comm_size(MPI_Comm comm, int *size)</pre>
14 15	<pre>int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)</pre>
16	<pre>int MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>
17 18	<pre>int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)</pre>
19 20	int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
ticket140. $\frac{21}{22}$	int MPI_Group_excl(MPI_Group group, int n, const int *ranks, MPI_Group *newgroup)
24	<pre>int MPI_Group_free(MPI_Group *group)</pre>
ticket140. $\frac{^{25}}{_{26}}$	int MPI_Group_incl(MPI_Group group, int n, const int *ranks, MPI_Group *newgroup)
28 29	<pre>int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>
30 31 32	<pre>int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>
33 34	<pre>int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>
35 36	<pre>int MPI_Group_rank(MPI_Group group, int *rank)</pre>
37	<pre>int MPI_Group_size(MPI_Group group, int *size)</pre>
ticket140. 39 40	int MPI_Group_translate_ranks (MPI_Group group1, int n, const int *ranks1, MPI_Group group2, int *ranks2)
41 42	int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
43 44 45 46 47 48	<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)</pre>

int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)	1 2
<pre>int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	3 4 5
<pre>int *type_keyval, void *extra_state)</pre>	6
<pre>int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)</pre>	$^{7}_{8}$ ticket252-W.
<pre>int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	9 10 11
<pre>int MPI_Type_free_keyval(int *type_keyval)</pre>	12
<pre>int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val, int *flag)</pre>	$^{13}_{14}$ ticket252-W.
<pre>int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int</pre>	$^{16}_{17}$ ticket252-W.
<pre>int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,</pre>	19 20 21
<pre>int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void *attribute_val, void *extra_state)</pre>	$^{22}_{23}$ ticket252-W.
<pre>int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val)</pre>	25 ticket252-W.
int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)	27 ticket252-W.
<pre>int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval, void *extra_state)</pre>	²⁸ ticket140. ²⁹ ³⁰
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)	32
int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)	33 34 35
int MPI_Win_free_keyval(int *win_keyval)	36
<pre>int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>	37 38 39
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)	40
int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state,	41 42 43
void *attribute_val_in, void *attribute_val_out, int *flag) int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void *attribute_val, void *extra_state)	44 45
<pre>int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)</pre>	46 47 48

ticket 140. $^{1}_{2}$	<pre>int MPI_Win_set_name(MPI_Win win, const char *win_name)</pre>
3 4	A.2.5 Process Topologies C Bindings
ticket125. $\frac{6}{7}$	<pre>int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,</pre>
ticket140. s ticket126. 9 ticket140. 10	<pre>int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int [*dims]dims[],</pre>
ticket126. $_{11}$	<pre>int MPI_Cartdim_get(MPI_Comm comm, int *ndims)</pre>
ticket125. 13 ticket125. 14	<pre>int MPI_Cart_get(MPI_Comm comm, int maxdims, int [*dims]dims[],</pre>
$ticket 125{15}$ $ticket 140{16}$ $ticket 126{17}$	<pre>int MPI_Cart_map(MPI_Comm comm, int ndims, const int [*dims]dims[], const</pre>
$\frac{\text{ticket 140.}_{18}}{\text{ticket 140.}_{19}}$	<pre>int MPI_Cart_rank(MPI_Comm comm, const int [*coords]coords[], int *rank)</pre>
ticket126. $\frac{19}{20}$	<pre>int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>
ticket140. 22 ticket126. 23	<pre>int MPI_Cart_sub(MPI_Comm comm, const int [*remain_dims]remain_dims[],</pre>
24 25	<pre>int MPI_Dims_create(int nnodes, int ndims, int *dims)</pre>
ticket140. ²⁶ ticket140. ²⁷ ticket140. ²⁸ ticket140. ²⁹	<pre>int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree, const</pre>
30 ticket140. 31 ticket140. 32 ticket140. 32 ticket140. 33 ticket140. 34	<pre>int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>
35 36	<pre>int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>
37 38 39 40	<pre>int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>
ticket140. 41 ticket125. 42 ticket140. 43	<pre>int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const</pre>
ticket 125. 44 45 46 ticket 125. 47 ticket 126. 48	<pre>int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges) int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,</pre>

int	<pre>MPI_Graph_map(MPI_Comm comm, int nnodes, const int [*index]index[],</pre>	¹ ticket140. ² ticket126.
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	³ ticket140. 4 ticket126.
int	<pre>MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	$\frac{5}{6}$ ticket125.
int	<pre>MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,</pre>	⁷ / ₈ ticket140. 9
int	<pre>MPI_Ineighbor_allgatherv(const void* sendbuf, int sendcount,</pre>	¹¹ ticket140. ¹² ticket140. ¹³ ticket140. ¹⁴
int	<pre>MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	 ¹⁶ ticket140. ¹⁷ ¹⁸
int	<pre>MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>	 ¹⁹ ticket140. ²⁰ ticket140. ²¹ ticket140. ²² ticket140. ²³ ticket140.
int	<pre>MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>	 ²⁴ ticket140. ²⁵ ticket140. ²⁶ ticket140. ²⁷ ticket299. ²⁸ ticket140.
int	<pre>MPI_Neighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>	²⁹ ticket140. ticket140. ticket299. ³¹ ticket140.
int	<pre>MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount,</pre>	 ³² ticket140. ³³ ticket140. ³⁴ ticket140. ³⁵ ticket140.
int	<pre>MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>	³⁶ ₃₇ ticket140. ³⁸ ₃₉
int	<pre>MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>	 40 ticket140. 41 ticket140. 42 ticket140. 43 ticket140. 44 ticket140.
int	<pre>MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>	⁴⁵ ticket140. ticket140. ticket140. ⁴⁷ ticket299. ⁴⁸ ticket140.
	Unofficial Draft for Comment Only	ticket140. ticket140.

ticket299. ticket140.

	1	<pre>int MPI_Topo_test(MPI_Comm comm, int *status)</pre>	
	3		
	4	A.2.6 MPI Environmental Management C Bindings	
	5 6	<pre>int MPI_Abort(MPI_Comm comm, int errorcode)</pre>	
	7	<pre>int MPI_Add_error_class(int *errorclass)</pre>	
	8 9	<pre>int MPI_Add_error_code(int errorclass, int *errorcode)</pre>	
ticket140.		<pre>int MPI_Add_error_string(int errorcode, const char *string)</pre>	
	11 12	int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)	
	12	<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)</pre>	
ticket252-W.	14 15 16	<pre>int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *[function]comm_errhandler_fn, MPI_Errhandler *errhandler)</pre>	
	17	int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)	
	18 19	<pre>int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)</pre>	
	20	int MPI_Errhandler_free(MPI_Errhandler *errhandler)	
	21 22	<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>	
	23	<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>	
	24 25	<pre>int MPI_File_call_errhandler(MPI_File fh, int errorcode)</pre>	
ticket252-W.		<pre>int MPI_File_create_errhandler(MPI_File_errhandler_function *[function]file_errhandler_fn, MPI_Errhandler *errhandler)</pre>	
	28 29	<pre>int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)</pre>	
	30 31	<pre>int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)</pre>	
	32	<pre>int MPI_Finalized(int *flag)</pre>	
	33	<pre>int MPI_Finalize(void)</pre>	
	34 35	<pre>int MPI_Free_mem(void *base)</pre>	
	36 37	<pre>int MPI_Get_library_version(char *version, int *resultlen)</pre>	
	38	<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>	
	39 40	<pre>int MPI_Get_version(int *version, int *subversion)</pre>	
	41	int MPI_Initialized(int *flag)	
	42 43	<pre>int MPI_Init(int *argc, char ***argv)</pre>	
	44	int MPI_Win_call_errhandler(MPI_Win win, int errorcode)	
	45 46	<pre>int MPI_Win_create_errhandler(MPI_Win_errhandler_function</pre>	
ticket252-W.		<pre>*[function]win_errhandler_fn, MPI_Errhandler *errhandler)</pre>	
	48	<pre>int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)</pre>	

int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)	1
double MPI_Wtick(void)	2 3
double MPI_Wtime(void)	4
	5
	6
A.2.7 The Info Object C Bindings	7
<pre>int MPI_Info_create(MPI_Info *info)</pre>	8 9
<pre>int MPI_Info_delete(MPI_Info info, const char *key)</pre>	⁹ 10 ticket140.
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)	11 12
<pre>int MPI_Info_free(MPI_Info *info)</pre>	13
<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>	$^{14}_{15}$ ticket140.
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)	17
	18
<pre>int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)</pre>	19
<pre>int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,</pre>	20 ticket140.
<pre>int MPI_Info_set(MPI_Info info, const char *key, const char *value)</pre>	$^{22}_{23}$ ticket140. $^{24}_{24}$ ticket140.
A.2.8 Process Creation and Management C Bindings	25
A.2.8 Process Creation and Management C Bindings int MPI_Close_port(const char *port_name)	
	25 26
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. 28 29 ticket140.
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. ²⁸ ticket140. 29 30 31 ticket140. 32 33
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. ²⁸ ticket140. 29 30 31 ticket140. 32 33 34
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. ²⁸ ticket140. 29 30 31 ticket140. 32 33
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. ²⁸ ticket140. 29 30 31 ticket140. 32 33 34 35
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. ²⁸ ticket140. 29 30 31 ticket140. 32 33 34 35 36
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. 28 ticket140. 29 30 31 ticket140. 32 33 34 35 36 37 38 ticket140. 39
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. 28 ticket140. 29 30 31 ticket140. 32 33 34 35 36 37 38 ticket140. 39
<pre>int MPI_Close_port(const char *port_name) int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	25 26 27 ticket140. 28 ticket140. 29 30 31 ticket140. 32 33 34 35 36 37 38 ticket140. 39

1	<pre>int MPI_Open_port(MPI_Info info, char *port_name)</pre>
$\operatorname{ticket140.}_{4}^{2}$ ticket140.	<pre>int MPI_Publish_name(const char *service_name, MPI_Info info, const</pre>
ticket140. 5 ticket140. 6 7 8	<pre>int MPI_Unpublish_name(const char *service_name, MPI_Info info, const</pre>
9	A.2.9 One-Sided Communications C Bindings
ticket140. 10 11 12 13 14	<pre>int MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)</pre>
ticket140a. 15 ticket140a. 16 17	<pre>int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)</pre>
ticket140a. $\frac{18}{19}$	<pre>int MPI_Fetch_and_op(const void *origin_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win)</pre>
ticket140a. 22 23 24 25 26	<pre>int MPI_Get_accumulate(const void *origin_addr, int origin_count,</pre>
27 28 29 30 31	int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
ticket140. 32 33 34	<pre>int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype</pre>
ticket140a. 35 36 37 38 39 40	<pre>int MPI_Raccumulate(const void *origin_addr, int origin_count,</pre>
ticket140a. 41 42 43 44 45 46 47	<pre>int MPI_Rget_accumulate(const void *origin_addr, int origin_count,</pre>
47	<pre>int MPI_Rget(void *origin_addr, int origin_count,</pre>

MPI_Datatype origin_datatype, int target_rank,	1 2
MPI_Aint target_disp, int target_count,	3
MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)	4
	5
int MPI_Rput(const void *origin_addr, int origin_count,	$_{6}$ ticket140a.
<pre>MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,</pre>	7
MPI_Datatype target_datatype, MPI_Win win,	8
MPI_Request *request)	9 10
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,	11
MPI_Comm comm, void *baseptr, MPI_Win *win)	12
	13
int MPI_Win_allocate_shared(MPI_Aint size, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)	14
	15 16
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)	17
<pre>int MPI_Win_complete(MPI_Win win)</pre>	18
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)	19
	20
<pre>int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)</pre>	21 22
	23
<pre>int MPI_Win_detach(MPI_Win win, const void *base)</pre>	$^{20}_{24}$ ticket 140a.
int MPI_Win_fence(int assert, MPI_Win win)	
int in i_win_tence(int assert, in i_win win)	25
int MPI_Win_flush_all(MPI_Win win)	25 26 27
	26
<pre>int MPI_Win_flush_all(MPI_Win win)</pre>	26 27
int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win)	26 27 28 29
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win)</pre>	26 27 28 29 30 31 32
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win)</pre>	26 27 28 29 30 31
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group)</pre>	26 27 28 29 30 31 32 33
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win)</pre>	26 27 28 29 30 31 32 33 34 35 36
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group)</pre>	26 27 28 29 30 31 32 33 34 35 36 37
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win)</pre>	26 27 28 29 30 31 32 33 34 35 36
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
<pre>int MPI_Win_flush_all(MPI_Win win) int MPI_Win_flush(int rank, MPI_Win win) int MPI_Win_flush_local_all(MPI_Win win) int MPI_Win_flush_local(int rank, MPI_Win win) int MPI_Win_free(MPI_Win *win) int MPI_Win_get_group(MPI_Win win, MPI_Group *group) int MPI_Win_lock_all(int assert, MPI_Win win) int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

```
1
              int MPI_Win_unlock(int rank, MPI_Win win)
         \mathbf{2}
              int MPI_Win_wait(MPI_Win win)
         3
         4
         \mathbf{5}
              A.2.10 External Interfaces C Bindings
         6
              int MPI_Grequest_complete(MPI_Request request)
         7
         8
              int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
         9
                            MPI_Grequest_free_function *free_fn,
         10
                            MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
         11
                            MPI_Request *request)
        12
              int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
        13
                            int *provided)
        14
         15
              int MPI_Is_thread_main(int *flag)
        16
        17
              int MPI_Query_thread(int *provided)
        18
              int MPI_Status_set_cancelled(MPI_Status *status, int flag)
        19
        20
              int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
                            int count)
        21
        22
              int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
        23
                            MPI_Count count)
        ^{24}
        25
        26
              A.2.11 I/O C Bindings
        27
              int MPI_File_close(MPI_File *fh)
        28
ticket 140. ^{29}
              int MPI_File_delete(const char *filename, MPI_Info info)
         30
              int MPI_File_get_amode(MPI_File fh, int *amode)
        ^{31}
        32
              int MPI_File_get_atomicity(MPI_File fh, int *flag)
        33
              int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
        34
                            MPI_Offset *disp)
        35
        36
              int MPI_File_get_group(MPI_File fh, MPI_Group *group)
        37
              int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
        38
        39
              int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
        40
        41
              int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
        42
              int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
        43
        44
              int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
        45
                            MPI_Aint *extent)
        46
              int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
        47
                            MPI_Datatype *filetype, char *datarep)
        48
```

int	<pre>MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,</pre>	1 2
int	<pre>MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	3 4 5
int	<pre>MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	6 7 8
int	<pre>MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	⁹ ticket140.
int	<pre>MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	¹¹ ticket140.
int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	¹⁰ ₁₄ ticket140.
int	<pre>MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)</pre>	¹⁶ ticket140.
int	MPI_File_preallocate(MPI_File fh, MPI_Offset size)	19
int	<pre>MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	20 21 22
int	<pre>MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	23
int	<pre>MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	24 25 26
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	27 28
int	<pre>MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	29 30
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	31 32
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	33 34 35
int	<pre>MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	36 37
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	38 39 40
int	MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)	41
int	MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	42 43 44
int	<pre>MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	45 46
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	47 48

754

ANNEX A. LANGUAGE BINDINGS SUMMARY

1	<pre>int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)</pre>
2 3	<pre>int MPI_File_set_atomicity(MPI_File fh, int flag)</pre>
4	<pre>int MPI_File_set_info(MPI_File fh, MPI_Info info)</pre>
5 6	int MPI_File_set_size(MPI_File fh, MPI_Offset size)
7 ticket140. ⁸	int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)
9 10	<pre>int MPI_File_sync(MPI_File fh)</pre>
ticket140. $\frac{11}{12}$	int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)
ticket140. 14	<pre>int MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
ticket140. $\frac{16}{17}$	int MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
ticket140. 19 20	<pre>int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, const</pre>
ticket140. $\frac{^{21}}{^{22}}$	<pre>int MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
ticket140. 24 25	<pre>int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
ticket140. $\frac{^{26}}{_{27}}$	int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
ticket140. 29 30	<pre>int MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
ticket140. $\frac{^{31}}{^{32}}$	int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)
ticket140. ³⁴ ³⁵	<pre>int MPI_File_write_ordered_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
ticket140. ³⁶ 37 38	int MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
ticket140. ³⁹ 40	int MPI_File_write_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
41 ticket140. 42 43 44 45 46 47	<pre>int MPI_Register_datarep(const char *datarep,</pre>
48	

A.2.12 Language Bindings C Bindings	1
int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)	2 3
int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)	4
	5
<pre>int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)</pre>	6
<pre>int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)</pre>	7 8
<pre>int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)</pre>	9
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)	$^{10}_{11}$ ticket252-W.
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)	12
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)	13
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	14 15
	16
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	17
MPI_Fint MPI_File_c2f(MPI_File file)	18
MPI_File MPI_File_f2c(MPI_Fint file)	19 20
MPI_Fint MPI_Group_c2f(MPI_Group group)	21
MPI_Group MPI_Group_f2c(MPI_Fint group)	22
MPI_Fint MPI_Info_c2f(MPI_Info info)	23 24
	25
MPI_Info MPI_Info_f2c(MPI_Fint info)	26
MPI_Fint MPI_Message_c2f(MPI_Message message)	27 28
MPI_Message MPI_Message_f2c(MPI_Fint message)	29
MPI_Fint MPI_Op_c2f(MPI_Op op)	30
MPI_Op MPI_Op_f2c(MPI_Fint op)	31 32
MPI_Fint MPI_Request_c2f(MPI_Request request)	33
MPI_Request MPI_Request_f2c(MPI_Fint request)	34
	35
<pre>int MPI_Status_c2f08(const MPI_Status *c_status, MPI_F08_status</pre>	$^{36}_{37}$ ticket140.
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)	38 39 ticket140.
int MPI_Status_f082c(const MPI_F08_status *f08_status, MPI_Status	40 ticket 140.
*c_status)	41
<pre>int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)</pre>	42 43 ticket140.
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)	44
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)	45 46
	47
MPI_Fint MPI_Win_c2f(MPI_Win win)	48

	1	MPI_Win MPI_Win_f2c(MPI_Fint win)
	3 4	A.2.13 Profiling Interface C Bindings
	5 6	<pre>int MPI_Pcontrol(const int level,)</pre>
	7 8	A.2.14 Deprecated C Bindings
	9	int MPI_Address(void* location, MPI_Aint *address)
	10 11	int MPI_Attr_delete(MPI_Comm comm, int keyval)
	12	<pre>int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)</pre>
	13 14	<pre>int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)</pre>
	15	int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
	16 17	void *attribute_val_in, void *attribute_val_out, int *flag)
ticket252-W.	19	<pre>int MPI_Errhandler_create(MPI_Handler_function *[function]handler_fn,</pre>
	20 21	int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
	22	int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
	23 24 25	<pre>int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn, int *keyval, void* extra_state)</pre>
	26	int MPI_Keyval_free(int *keyval)
	27 28	int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
	29	void *attribute_val_in, void *attribute_val_out, int *flag)
	30 31	<pre>int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state)</pre>
	32 33	int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
	34	int MPI_Type_hindexed(int count, int *array_of_blocklengths,
	35 36	MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
	37	MPI_Datatype *newtype)
	38 39	<pre>int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,</pre>
	40	
	41	<pre>int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)</pre>
	42 43	<pre>int MPI_Type_struct(int count, int *array_of_blocklengths,</pre>
	44	MPI_Datatype *array_of_types, MPI_Datatype *newtype)
	45 46	<pre>int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)</pre>
	47	
	48	

	$\frac{1}{2}$ ticket247-
A.3 Fortran 2008 Bindings with the mpi_f08 Module	3 4
A.3.1 Point-to-Point Communication Fortran 2008 Bindings	5 ticket-248
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)	7
TYPE(*), DIMENSION(), INTENT(IN) :: buf	8
INTEGER, INTENT(IN) :: count, dest, tag	9
TYPE(MPI_Datatype), INTENT(IN) :: datatype	10
TYPE(MPI_Comm), INTENT(IN) :: comm	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)	$_{13}$ ticket-248
BIND(C)	14
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	15
INTEGER, INTENT(IN) :: count, dest, tag	16
TYPE(MPI_Datatype), INTENT(IN) :: datatype	17
TYPE(MPI_Comm), INTENT(IN) :: comm	18
TYPE(MPI_Request), INTENT(OUT) :: request	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21 ticket-248
MPI_Buffer_attach(buffer, size, ierror) BIND(C)	22
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer INTEGER, INTENT(IN) :: size	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
INTEGER, OFIIONAL, INTENI(001) IEIIOI	$\frac{1}{26}$ ticket-248
MPI_Buffer_detach(buffer_addr, size, ierror) BIND(C)	27
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	28
TYPE(C_PTR), INTENT(OUT) :: buffer_addr	29
INTEGER, INTENT(OUT) :: size	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31 ticket-248
MPI_Cancel(request, ierror) BIND(C)	32
TYPE(MPI_Request), INTENT(IN) :: request	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
	$_{35}$ ticket-248
MPI_Get_count(status, datatype, count, ierror) BIND(C)	36
TYPE(MPI_Status), INTENT(IN) :: status TYPE(MPI_Datatype), INTENT(IN) :: datatype	37
INTEGER, INTENT(OUT) :: count	38
INTEGER, INTENI(001) :: Count INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
INTEGER, OFFICIAL, INTENT(UOT) ICITOL	40 ticket-248
<pre>MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)</pre>	41
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	42
INTEGER, INTENT(IN) :: count, dest, tag	43
TYPE(MPI_Datatype), INTENT(IN) :: datatype	44
TYPE(MPI_Comm), INTENT(IN) :: comm	45 46
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46 47
	4±1

1	MPT Improbe(gource tog comm flog meggage status ierror) RIND(C)	
2	<pre>MPI_Improbe(source, tag, comm, flag, message, status, ierror) BIND(C) INTEGER, INTENT(IN) :: source, tag</pre>	
3	TYPE(MPI_Comm), INTENT(IN) :: comm	
4	INTEGER, INTENT(OUT) :: flag	
5	TYPE(MPI_Message), INTENT(OUT) :: message	
6	TYPE(MPI_Status) :: status	
7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
ticket-248 T. $_{\rm 8}$	INTEGER, OFFICIAL, INTENT(COT) TETTOT	
9	<pre>MPI_Imrecv(buf, count, datatype, message, request, ierror) BIND(C)</pre>	
10	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	
11	INTEGER, INTENT(IN) :: count	
12	TYPE(MPI_Datatype), INTENT(IN) :: datatype	
13	TYPE(MPI_Message), INTENT(INOUT) :: message	
14	TYPE(MPI_Request), INTENT(OUT) :: request	
ticket-248T. ¹⁵	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
16 ticket-2401.	MPI_Iprobe(source, tag, comm, flag, status, ierror) BIND(C)	
17	INTEGER, INTENT(IN) :: source, tag	
18	TYPE(MPI_Comm), INTENT(IN) :: comm	
19	LOGICAL, INTENT(OUT) :: flag	
20	TYPE(MPI_Status) :: status	
21	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
ticket-248T. $_{22}$		
23	MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND((3)
24	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	
25	INTEGER, INTENT(IN) :: count, source, tag	
26	TYPE(MPI_Datatype), INTENT(IN) :: datatype	
27	TYPE(MPI_Comm), INTENT(IN) :: comm	
28	TYPE(MPI_Request), INTENT(OUT) :: request	
ticket-248T. ²⁹	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
30	MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C))
31	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	
32	INTEGER, INTENT(IN) :: count, dest, tag	
33	TYPE(MPI_Datatype), INTENT(IN) :: datatype	
34	TYPE(MPI_Comm), INTENT(IN) :: comm	
35	TYPE(MPI_Request), INTENT(OUT) :: request	
36	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
ticket-248T. 37		
38	MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)	
39	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	
40	INTEGER, INTENT(IN) :: count, dest, tag	
41	TYPE(MPI_Datatype), INTENT(IN) :: datatype	
42	TYPE(MPI_Comm), INTENT(IN) :: comm	
43	TYPE(MPI_Request), INTENT(OUT) :: request	
ticket-248T. 44	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
45	MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C))
46	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	
47	INTEGER, INTENT(IN) :: count, dest, tag	
48	-	

TYPE(MPI_Datatype), INTENT(IN) :: datatype	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2
TYPE(MPI_Request), INTENT(OUT) :: request	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	$_5$ ticket-248T.
MPI_Mprobe(source, tag, comm, message, status, ierror) BIND(C)	6
INTEGER, INTENT(IN) :: source, tag	7
TYPE(MPI_Comm), INTENT(IN) :: comm	8
TYPE(MPI_Message), INTENT(OUT) :: message	9
TYPE(MPI_Status) :: status	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11 ticket-248T.
MPI_Mrecv(buf, count, datatype, message, status, ierror) BIND(C)	12 ticket-2401.
TYPE(*), DIMENSION() :: buf	13
INTEGER, INTENT(IN) :: count	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
TYPE(MFI_Datatype), INTENT(IN) datatype TYPE(MPI_Message), INTENT(INOUT) :: message	16
TYPE(MPI_Message), INTENT(INUOT) :: message TYPE(MPI_Status) :: status	17
	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	¹⁹ ticket-248T.
MPI_Probe(source, tag, comm, status, ierror) BIND(C)	20
INTEGER, INTENT(IN) :: source, tag	20
TYPE(MPI_Comm), INTENT(IN) :: comm	22
TYPE(MPI_Status) :: status	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
,,,,	24 ticket-248T.
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C)	25
TYPE(*), DIMENSION() :: buf	25 26
	25 26 27
TYPE(*), DIMENSION() :: buf	25 26 27 28
TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag	25 26 27 28 29
TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype	25 26 27 28
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	25 26 27 28 29 30 31
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	25 26 27 28 29 30
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre> MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)	25 26 27 28 29 30 31
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C)</pre>	25 26 27 28 29 30 31 32 ticket-248T.
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 40
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Request_free(request, ierror) BIND(C)</pre>	 25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 ⁴⁰ ticket-248T.
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Request_free(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(INOUT) :: request</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 ⁴⁰ ticket-248T. 41
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Request_free(request, ierror) BIND(C)</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 ⁴⁰ ticket-248T. 42
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Request_free(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(INOUT) :: request</pre>	25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 ⁴⁰ ticket-248T. ⁴¹ 42 43
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(*), DIMENSION(), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	 25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 40 41 42 43 44 ticket-248T.
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Request_free(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(INOUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	 25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 40 40 ticket-248T. 42 43 44 ticket-248T. 45
<pre>TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Request_free(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(INOUT) :: ierror MPI_Request_get_status(request, flag, status, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request</pre>	 25 26 27 28 29 30 31 32 ticket-248T. 33 34 35 36 37 38 39 40 40 41 42 43 44 ticket-248T. 45 46

ticket-248T.	1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UICKCU-2401.	-	I_Rsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
	4	TYPE(*), DIMENSION(), INTENT(IN) :: buf
	5	INTEGER, INTENT(IN) :: count, dest, tag
	6	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	7	TYPE(MPI_Comm), INTENT(IN) :: comm
ticket-248T.	8	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UCKEU-2401.	⁹ MP	I_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
	10	BIND(C)
	11	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	12	INTEGER, INTENT(IN) :: count, dest, tag
	13	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	14	TYPE(MPI_Comm), INTENT(IN) :: comm
	15	TYPE(MPI_Request), INTENT(OUT) :: request
ticket-248T.	16	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	MD	I_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
		TYPE(*), DIMENSION(), INTENT(IN) :: buf
	19 20	INTEGER, INTENT(IN) :: count, dest, tag
	20 21	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	22	TYPE(MPI_Comm), INTENT(IN) :: comm
	23	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-2481.	~ (T Condinit/buf court determine dont ton communications)
	25 1912	<pre>'I_Send_init(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)</pre>
	26	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	27	INTEGER, INTENT(IN) :: count, dest, tag
	28	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	29	TYPE(MPI_Comm), INTENT(IN) :: comm
	30	TYPE(MPI_Request), INTENT(OUT) :: request
	31	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.		
		I_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
	34	<pre>comm, status, ierror) BIND(C) TYPE(*), DIMENSION() :: buf</pre>
	35	INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
	36	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	37 38	TYPE(MPI_Comm), INTENT(IN) :: comm
	39	TYPE(MPI_Status) :: status
	40	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	41	T Conducer (conduct conduct dest conduct to the second for
	42 MP	I_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
	43	recvcount, recvtype, source, recvtag, comm, status, ierror) BIND(C)
	44	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf
	45	TYPE(*), DIMENSION() :: recvbuf
	46	INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
	47	recvtag
	48	

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   3
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
                                                                                   _{5} ticket-248T.
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
                                                                                   6
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   7
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                   8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   9
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   11
                                                                                     ticket-248T.
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                   12
                                                                                   13
              BIND(C)
                                                                                   14
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                   15
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                   16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   17
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   <sub>20</sub> ticket-248T.
MPI_Startall(count, array_of_requests, ierror) BIND(C)
                                                                                   21
    INTEGER, INTENT(IN) :: count
                                                                                   22
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   ^{24} ticket-248T.
                                                                                   25
MPI_Start(request, ierror) BIND(C)
                                                                                   26
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   <sub>28</sub> ticket-248T.
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
                                                                                   29
              BIND(C)
                                                                                   30
    INTEGER, INTENT(IN) :: count
                                                                                   31
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                   32
    LOGICAL, INTENT(OUT) :: flag
                                                                                   33
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                   34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   35
                                                                                     ticket-248T.
                                                                                   36
MPI_Testany(count, array_of_requests, index, flag, status, ierror) BIND(C)
                                                                                   37
    INTEGER, INTENT(IN) :: count
                                                                                   38
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                   39
    INTEGER, INTENT(OUT) :: index
                                                                                   40
    LOGICAL, INTENT(OUT) :: flag
                                                                                   41
    TYPE(MPI_Status) :: status
                                                                                   42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   <sup>43</sup> ticket-248T.
MPI_Test_cancelled(status, flag, ierror) BIND(C)
                                                                                   44
    TYPE(MPI_Status), INTENT(IN) :: status
                                                                                   45
    LOGICAL, INTENT(OUT) :: flag
                                                                                   46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   _{48} ticket-248T.
```

	1	<pre>MPI_Test(request, flag, status, ierror) BIND(C)</pre>
	2	TYPE(MPI_Request), INTENT(INOUT) :: request
	3	LOGICAL, INTENT(OUT) :: flag
	4	TYPE(MPI_Status) :: status
· 1 · 0.40m	5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	6	MDI Testasme (incount error of requests outcount error of indices
	7	MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
	8	array_of_statuses, ierror) BIND(C) INTEGER, INTENT(IN) :: incount
	9	
	10	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount) INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
	11	
	12	TYPE(MPI_Status) :: array_of_statuses(*)
ticket-248T.	13	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10100 - 10 - 1	14	<pre>MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) BIND(C)</pre>
	15	INTEGER, INTENT(IN) :: count
	16	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
	17	TYPE(MPI_Status) :: array_of_statuses(*)
	18	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	19	
	20	<pre>MPI_Waitany(count, array_of_requests, index, status, ierror) BIND(C)</pre>
	21	INTEGER, INTENT(IN) :: count
	22	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
	23	INTEGER, INTENT(OUT) :: index
	24	TYPE(MPI_Status) :: status
ticket-248T.	25	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-2401.	26	MPI_Wait(request, status, ierror) BIND(C)
	27	TYPE(MPI_Request), INTENT(INOUT) :: request
	28	TYPE(MPI_Status) :: status
	29	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	30	
	31	<pre>MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,</pre>
	32	array_of_statuses, ierror) BIND(C)
	33	INTEGER, INTENT(IN) :: incount
	34	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
	35	<pre>INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)</pre>
	36	<pre>TYPE(MPI_Status) :: array_of_statuses(*)</pre>
	37	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	38	
	39	A 2 2 Detetures Foutron 2008 Findings
ticket-248T.	40	A.3.2 Datatypes Fortran 2008 Bindings
	41	MPI_Get_address(location, address, ierror) BIND(C)
	42	TYPE(*), DIMENSION(), ASYNCHRONOUS :: location
	43	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
	44	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	45	
	46	MPI_Get_elements(status, datatype, count, ierror) BIND(C)
	47	TYPE(MPI_Status), INTENT(IN) :: status
	48	TYPE(MPI_Datatype), INTENT(IN) :: datatype

INTEGER, INTENT(OUT) :: count	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2 +:-l+ 040T
MPI_Get_elements_x(status, datatype, count, ierror) BIND(C)	$_{3}$ ticket-248T.
••	4
TYPE(MPI_Status), INTENT(IN) :: status	5
TYPE(MPI_Datatype), INTENT(IN) :: datatype	6
INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: count	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8 ticket-248T.
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,	9
position, ierror) BIND(C)	10
CHARACTER(LEN=*), INTENT(IN) :: datarep	11
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	12
TYPE(*), DIMENSION() :: outbuf	13
INTEGER, INTENT(IN) :: incount	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize	16
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) Outsize INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position	17
INTEGER (NIND-MPI_ADDRESS_KIND), INTENT(INDOI) :: position INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
INIEGER, UPIIUNAL, INIENI(UUI) :: lerror	¹⁹ ticket-248T.
MPI_Pack_external_size(datarep, incount, datatype, size, ierror) BIND(C)	20
TYPE(MPI_Datatype), INTENT(IN) :: datatype	20
INTEGER, INTENT(IN) :: incount	22
CHARACTER(LEN=*), INTENT(IN) :: datarep	23
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	$^{25}_{26}$ ticket-248T.
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)	
BIND(C)	27
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	28
TYPE(*), DIMENSION() :: outbuf	29
INTEGER, INTENT(IN) :: incount, outsize	30
TYPE(MPI_Datatype), INTENT(IN) :: datatype	31
INTEGER, INTENT(INOUT) :: position	32
TYPE(MPI_Comm), INTENT(IN) :: comm	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MDT Deele size (in sound, detetune, some size, issues) DIND(0)	35 ticket-248T.
MPI_Pack_size(incount, datatype, comm, size, ierror) BIND(C)	36
INTEGER, INTENT(IN) :: incount	37
TYPE(MPI_Datatype), INTENT(IN) :: datatype	38
TYPE(MPI_Comm), INTENT(IN) :: comm	39
INTEGER, INTENT(OUT) :: size	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	⁴¹ ticket-248T.
MPI_Type_commit(datatype, ierror) BIND(C)	42
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
,, ,, , ,, , ,, , ,, , ,, , , , , , , , , , , , , , , , , , , ,	$_{45}$ ticket-248T.
MPI_Type_contiguous(count, oldtype, newtype, ierror) BIND(C)	46
INTEGER, INTENT(IN) :: count	47
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	48

	1	
		TYPE(MPI_Datatype), INTENT(OUT) :: newtype
ticket-248T.	2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
01CIKC0 2401.	Č.	MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
	4	array_of_distribs, array_of_dargs, array_of_psizes, order,
	5	oldtype, newtype, ierror) BIND(C)
	6	INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
	7	array_of_distribs(ndims), array_of_dargs(ndims),
	8	array_of_psizes(ndims), order
	9	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	10	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
	11	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	12	INTEGER, OFITOWAL, INTENT(001) TETTOT
	13	<pre>MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,</pre>
	14	oldtype, newtype, ierror) BIND(C)
	15	INTEGER, INTENT(IN) :: count, blocklength
	16	INTEGER(KIND=MPI_Address_kind), INTENT(IN) ::
	17	array_of_displacements(count)
	18	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	19	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
	20	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	21	
	22	<pre>MPI_Type_create_hindexed(count, array_of_blocklengths,</pre>
	23	array_of_displacements, oldtype, newtype, ierror) BIND(C)
	24	<pre>INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)</pre>
	25	<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::</pre>
	26	<pre>array_of_displacements(count)</pre>
	27	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	28	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
	29	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	30	
	31	MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
	32	ierror) BIND(C)
	33	INTEGER, INTENT(IN) :: count, blocklength
	34	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
	35	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	36	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
ticket-248T.		INTEGER, OPTIONAL, INTENT(OUT) :: ierror
0101100 2 10 1	38	MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
	39	oldtype, newtype, ierror) BIND(C)
	40	INTEGER, INTENT(IN) :: count, blocklength,
	40	array_of_displacements(count)
		TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	42	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
	43	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.		INIDOLO, OFFICIAL, INIDAL(UOI/ ICILOI
	45	<pre>MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) BIND(C)</pre>
	46	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
	47	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	48	

TYPE(MPI_Datatype), INTENT(OUT) :: newtype	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2 tiplest 949T
MPI_Type_create_struct(count, array_of_blocklengths,	$_3$ ticket-248T.
array_of_displacements, array_of_types, newtype, ierror)	4
BIND(C)	5
	6
INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)	7
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::	8
array_of_displacements(count)	9
TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)	10
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12 ticket-248T.
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,	13
array_of_starts, order, oldtype, newtype, ierror) BIND(C)	14
INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),	15
array_of_subsizes(ndims), array_of_starts(ndims), order	16
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	17
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
	$_{20}$ ticket-248T.
MPI_Type_dup(oldtype, newtype, ierror) BIND(C)	21
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	22
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24 ticket-248T.
MPI_Type_free(datatype, ierror) BIND(C)	25 ticket-2401.
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
INTEGER, OFFICIARE, INTENT(001) TETTOT	²⁸ ticket-248T.
<pre>MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,</pre>	29
<pre>array_of_integers, array_of_addresses, array_of_datatypes,</pre>	30
array_of_integers, array_of_addresses, array_of_datatypes, ierror) BIND(C)	30 31
ierror) BIND(C)	31
ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype	31 32
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes</pre>	31 32 33
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers)</pre>	31 32 33 34
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::</pre>	31 32 33 34 35
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses)</pre>	31 32 33 34 35 36 37 38
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	31 32 33 34 35 36 37
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ ticket-248T.
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes, combiner, ierror) BIND(C)</pre>	31 32 33 34 35 36 37 ³⁸ 39 ticket-248T.
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ 39 ticket-248T. 39
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ 39 ticket-248T. 40 41
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ 39 ticket-248T. 39 40 41 42
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ ticket-248T. 39 40 41 42 43 44
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ ticket-248T. 39 40 41 42 43
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ ticket-248T. 39 40 41 42 43 44 45 ticket-248T.
<pre>ierror) BIND(C) TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes INTEGER, INTENT(OUT) :: array_of_integers(max_integers) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: array_of_addresses(max_addresses) TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes) INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,</pre>	31 32 33 34 35 36 37 ³⁸ ticket-248T. 40 41 42 43 44 45 ticket-248T. 46

ticket-248T.	1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UICKEU-2401.	-	MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) BIND(C)
	3	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	4	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
	5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.		
	7	<pre>MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,</pre>
	8	oldtype, newtype, ierror) BIND(C)
	9	<pre>INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),</pre>
	10 11	array_of_displacements(count)
	11	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
		TYPE(MPI_Datatype), INTENT(OUT) :: newtype
ticket-248T.	13	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	14 15	MPI_Type_size(datatype, size, ierror) BIND(C)
	15	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	10	INTEGER, INTENT(OUT) :: size
	17	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	19	
	20	MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
	20	BIND(C)
	21	INTEGER, INTENT(IN) :: count, blocklength, stride
	23	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	24	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
ticket-248T.		INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	26	MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
	27	datatype, ierror) BIND(C)
	28	CHARACTER(LEN=*), INTENT(IN) :: datarep
	29	TYPE(*), DIMENSION(), INTENT(IN) :: inbuf
	30	TYPE(*), DIMENSION() :: outbuf
	31	<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize</pre>
	32	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
	33	INTEGER, INTENT(IN) :: outcount
	34	TYPE(MPI_Datatype), INTENT(IN) :: datatype
tial of 940m	35	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	36	MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
	37	ierror) BIND(C)
	38	TYPE(*), DIMENSION(), INTENT(IN) :: inbuf
	39	TYPE(*), DIMENSION(), INTENT(IN) INDUI TYPE(*), DIMENSION() :: outbuf
	40	INTEGER, INTENT(IN) :: insize, outcount
	41	INTEGER, INTENT(IN) INSIZE, Outcount INTEGER, INTENT(INOUT) :: position
	42	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	43	TYPE(MPI_Comm), INTENT(IN) :: comm
	44	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	45	Interior of Home, Interior, 101101
	46	
	47	A.3.3 Collective Communication Fortran 2008 Bindings
ticket-248T.	48	

```
1
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 2
             comm, ierror) BIND(C)
                                                                                 3
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 5
                                                                                 6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 7
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 8
                                                                                 _{9} ticket-248T.
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                 10
             recvtype, comm, ierror) BIND(C)
                                                                                 11
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 12
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 13
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                 14
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 ^{17} ticket-248T.
                                                                                 18
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
                                                                                 19
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 20
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 21
    INTEGER, INTENT(IN) :: count
                                                                                 22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 23
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 26 ticket-248T.
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 27
             comm, ierror) BIND(C)
                                                                                 28
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 29
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 30
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 31
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 34
                                                                                   ticket-248T.
                                                                                 35
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                 36
             rdispls, recvtype, comm, ierror) BIND(C)
                                                                                 37
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 38
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 39
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                 40
    rdispls(*)
                                                                                 41
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44 ticket-248T.
MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
                                                                                 45
             rdispls, recvtypes, comm, ierror) BIND(C)
                                                                                 46
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 47
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 48
```

767

```
1
                   INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
           2
                   rdispls(*)
           3
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
           4
                   TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
           5
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           6
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
               MPI_Barrier(comm, ierror) BIND(C)
           8
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           9
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. ^{10}
          11
               MPI_Bcast(buffer, count, datatype, root, comm, ierror) BIND(C)
          12
                   TYPE(*), DIMENSION(..) :: buffer
          13
                   INTEGER, INTENT(IN) :: count, root
          14
                   TYPE(MPI_Datatype), INTENT(IN) :: datatype
           15
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           16
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 17
                MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
           18
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
           19
                   TYPE(*), DIMENSION(..) :: recvbuf
          20
                   INTEGER, INTENT(IN) :: count
          21
                   TYPE(MPI_Datatype), INTENT(IN) :: datatype
          22
                   TYPE(MPI_Op), INTENT(IN) :: op
          23
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          24
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          25
ticket-248T.
           26
                MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
          27
                             root, comm, ierror) BIND(C)
          28
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          29
                   TYPE(*), DIMENSION(..) :: recvbuf
          30
                   INTEGER, INTENT(IN) :: sendcount, recvcount, root
          31
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           32
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           33
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 34
               MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
          35
                             recvtype, root, comm, ierror) BIND(C)
          36
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          37
                   TYPE(*), DIMENSION(..) :: recvbuf
          38
                   INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
          39
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           40
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          41
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          42
ticket-248T.
           43
               MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
          44
                             comm, request, ierror) BIND(C)
          45
                   TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
           46
                   TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
           47
                   INTEGER, INTENT(IN) :: sendcount, recvcount
           48
```

768

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
                                                                                  _{5} ticket-248T.
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  6
             recvtype, comm, request, ierror) BIND(C)
                                                                                  7
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  8
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  9
    INTEGER, INTENT(IN) :: sendcount
                                                                                  10
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  11
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 13
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 <sup>15</sup> ticket-248T.
                                                                                  16
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
                                                                                 17
             ierror) BIND(C)
                                                                                  18
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
                                                                                  20
                                                                                 21
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 22
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 26 ticket-248T.
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 27
             comm, request, ierror) BIND(C)
                                                                                 28
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 29
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  30
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 ^{35} ticket-248T.
                                                                                 36
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                 37
             rdispls, recvtype, comm, request, ierror) BIND(C)
                                                                                 38
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  40
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  41
    recvcounts(*), rdispls(*)
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  44
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46 ticket-248T.
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  47
             recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
                                                                                  48
```

1 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 2 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 3 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), 4 recvcounts(*), rdispls(*) 5TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*), 6 recvtypes(*) 7 TYPE(MPI_Comm), INTENT(IN) :: comm 8 TYPE(MPI_Request), INTENT(OUT) :: request 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 10 MPI_Ibarrier(comm, request, ierror) BIND(C) 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12TYPE(MPI_Request), INTENT(OUT) :: request 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 14 15MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C) 16TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer 17 INTEGER, INTENT(IN) :: count, root 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19 TYPE(MPI_Comm), INTENT(IN) :: comm 20TYPE(MPI_Request), INTENT(OUT) :: request 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 22 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) 23 BIND(C) 24TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 25TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 26INTEGER, INTENT(IN) :: count 27TYPE(MPI_Datatype), INTENT(IN) :: datatype 28TYPE(MPI_Op), INTENT(IN) :: op 29 TYPE(MPI_Comm), INTENT(IN) :: comm 30 TYPE(MPI_Request), INTENT(OUT) :: request 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 ticket-248T. 33 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 34 root, comm, request, ierror) BIND(C) 35 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 36 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 37 INTEGER, INTENT(IN) :: sendcount, recvcount, root 38 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 39 TYPE(MPI_Comm), INTENT(IN) :: comm 40 TYPE(MPI_Request), INTENT(OUT) :: request 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 42 MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 43 recvtype, root, comm, request, ierror) BIND(C) 44 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 45 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 46 INTEGER, INTENT(IN) :: sendcount, root 47 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) 48

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
                                                                                  _{5} ticket-248T.
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                  6
             request, ierror) BIND(C)
                                                                                  7
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  8
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  9
    INTEGER, INTENT(IN) :: recvcount
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  11
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 13
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 <sup>15</sup> ticket-248T.
                                                                                 16
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                 17
             request, ierror) BIND(C)
                                                                                 18
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                 20
                                                                                 21
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 22
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 26 ticket-248T.
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                 27
             ierror) BIND(C)
                                                                                 28
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 29
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  30
    INTEGER, INTENT(IN) :: count, root
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  32
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 34
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 36
                                                                                   ticket-248T.
                                                                                 37
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                 38
             BIND(C)
                                                                                 39
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  41
    INTEGER, INTENT(IN) :: count
                                                                                 42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  43
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  45
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 47 ticket-248T.
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  48
```

1 root, comm, request, ierror) BIND(C) 2 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 3 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 4 INTEGER, INTENT(IN) :: sendcount, recvcount, root 5TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 6 TYPE(MPI_Comm), INTENT(IN) :: comm 7 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 10 recvtype, root, comm, request, ierror) BIND(C) 11 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 12TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 13 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) 14 INTEGER, INTENT(IN) :: recvcount, root 15TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 16 TYPE(MPI_Comm), INTENT(IN) :: comm 17 TYPE(MPI_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 19 20MPI_Op_commutative(op, commute, ierror) BIND(C) 21TYPE(MPI_Op), INTENT(IN) :: op 22 LOGICAL, INTENT(OUT) :: commute 23 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 24 MPI_Op_create(user_fn, commute, op, ierror) BIND(C) 25PROCEDURE(MPI_User_function) :: user_fn 26LOGICAL, INTENT(IN) :: commute 27TYPE(MPI_Op), INTENT(OUT) :: op 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 29 30 MPI_Op_free(op, ierror) BIND(C) 31 TYPE(MPI_Op), INTENT(INOUT) :: op 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 33 MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C) 34 TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf 35 TYPE(*), DIMENSION(..) :: inoutbuf 36 INTEGER, INTENT(IN) :: count 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 TYPE(MPI_Op), INTENT(IN) :: op 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40ticket-248T. 41 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 42ierror) BIND(C) 43 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 44TYPE(*), DIMENSION(..) :: recvbuf 45INTEGER, INTENT(IN) :: recvcount 46TYPE(MPI_Datatype), INTENT(IN) :: datatype 47 TYPE(MPI_Op), INTENT(IN) :: op 48

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  1
                                                                                  2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  _{3} ticket-248T.
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                  4
             ierror) BIND(C)
                                                                                  5
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  6
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  7
    INTEGER, INTENT(IN) :: recvcounts(*)
                                                                                  8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  9
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 ^{12} ticket-248T.
                                                                                 13
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
                                                                                 14
             BIND(C)
                                                                                  15
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  16
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  17
    INTEGER, INTENT(IN) :: count, root
                                                                                  18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  19
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 _{22} ticket-248T.
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
                                                                                 23
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 24
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                 25
    INTEGER, INTENT(IN) :: count
                                                                                  26
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 27
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 ^{30} ticket-248T.
                                                                                  ^{31}
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 32
             root, comm, ierror) BIND(C)
                                                                                 33
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 34
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 35
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                 36
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 39 ticket-248T.
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                  40
             recvtype, root, comm, ierror) BIND(C)
                                                                                 41
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 42
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 43
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  47
                                                                                  48
```

```
1
                A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
ticket-248T. 2
                MPI_Comm_compare(comm1, comm2, result, ierror) BIND(C)
           3
                    TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
           4
                    INTEGER, INTENT(OUT) :: result
           5
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           \mathbf{6}
ticket-248T.
           7
                MPI_Comm_create(comm, group, newcomm, ierror) BIND(C)
           8
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           9
                    TYPE(MPI_Group), INTENT(IN) :: group
           10
                    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
           11
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 12
                MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
           13
                              extra_state, ierror) BIND(C)
          14
                    PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
          15
                    PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
          16
                    INTEGER, INTENT(OUT) :: comm_keyval
           17
                    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
           18
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. ^{19}
          20
                MPI_Comm_delete_attr(comm, comm_keyval, ierror) BIND(C)
          21
                    TYPE(MPI_Comm), INTENT(IN) :: comm
          22
                    INTEGER, INTENT(IN) :: comm_keyval
          23
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 24
                MPI_Comm_dup(comm, newcomm, ierror) BIND(C)
          25
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           26
                    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
          27
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          28
ticket-248T.
           29
                MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
          30
                              attribute_val_out, flag, ierror) BIND(C)
          31
                    TYPE(MPI_Comm), INTENT(IN) :: oldcomm
          32
                    INTEGER, INTENT(IN) :: comm_keyval
          33
                    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
          34
                    attribute_val_in
          35
                    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
          36
                    LOGICAL, INTENT(OUT) :: flag
          37
                    INTEGER, INTENT(OUT) :: ierror
ticket-248T. 38
                MPI_Comm_free(comm, ierror) BIND(C)
          39
                    TYPE(MPI_Comm), INTENT(INOUT) :: comm
          40
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           41
ticket-248T.
           42
               MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)
          43
                    INTEGER, INTENT(INOUT) :: comm_keyval
          44
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 45
               MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror) BIND(C)
          46
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           47
                    INTEGER, INTENT(IN) :: comm_keyval
           48
```

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val	1
LOGICAL, INTENT(OUT) :: flag	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
	$_4$ ticket-248T.
MPI_Comm_get_name(comm, comm_name, resultlen, ierror) BIND(C)	5
TYPE(MPI_Comm), INTENT(IN) :: comm	-
CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name	6
	7
INTEGER, INTENT(OUT) :: resultlen	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	⁹ ticket-248T.
MPI_Comm_group(comm, group, ierror) BIND(C)	10 UICKE 0-240 1.
	11
TYPE(MPI_Comm), INTENT(IN) :: comm	12
TYPE(MPI_Group), INTENT(OUT) :: group	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13 ticlet 949T
NDT CONN NULL CODY DV(-1) h	$_{14}$ ticket-248T.
MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,	15
<pre>attribute_val_out, flag, ierror) BIND(C)</pre>	16
TYPE(MPI_Comm), INTENT(IN) :: oldcomm	17
INTEGER, INTENT(IN) :: comm_keyval	18
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,	19
attribute_val_in	
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out	20
LOGICAL, INTENT(OUT) :: flag	21
	22
INTEGER, INTENT(OUT) :: ierror	23 ticket-248T.
MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,	24
ierror) BIND(C)	25
	26
TYPE(MPI_Comm), INTENT(IN) :: comm	27
INTEGER, INTENT(IN) :: comm_keyval	
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,</pre>	28
extra_state	29
INTEGER, INTENT(OUT) :: ierror	30
	31 ticket-248T.
<pre>MPI_Comm_rank(comm, rank, ierror) BIND(C)</pre>	32
TYPE(MPI_Comm), INTENT(IN) :: comm	33
INTEGER, INTENT(OUT) :: rank	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
	$^{35}_{36}$ ticket-248T.
MPI_Comm_remote_group(comm, group, ierror) BIND(C)	
TYPE(MPI_Comm), INTENT(IN) :: comm	37
TYPE(MPI_Group), INTENT(OUT) :: group	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
,,,,,	⁴⁰ ticket-248T.
MPI_Comm_remote_size(comm, size, ierror) BIND(C)	41
TYPE(MPI_Comm), INTENT(IN) :: comm	42
INTEGER, INTENT(OUT) :: size	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
	$\frac{44}{12}$ ticket-248T.
<pre>MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) BIND(C)</pre>	45
TYPE(MPI_Comm), INTENT(IN) :: comm	46
INTEGER, INTENT(IN) :: comm_keyval	47
	48

	1 INTECED (KIND-MDI ADDRESS KIND) INTENT(IN) · · · · · · · · · · · · · · · · · · ·
ticket-248T	 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	3
	4 MPI_Comm_set_name(comm, comm_name, ierror) BIND(C)
	5 TYPE(MPI_Comm), INTENT(IN) :: comm
	6 CHARACTER(LEN=*), INTENT(IN) :: comm_name
ticket-248T.	7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	⁸ MPI_Comm_size(comm, size, ierror) BIND(C)
	⁹ TYPE(MPI_Comm), INTENT(IN) :: comm
	¹⁰ INTEGER, INTENT(OUT) :: size
ticket-248T.	¹¹ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UCKC0-2401.	¹² MPI_Comm_split(comm, color, key, newcomm, ierror) BIND(C)
	TYPE(MPI_Comm), INTENT(IN) :: comm
	15 INTEGER, INTENT(IN) :: color, key
	16 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
· 1 · 0400	17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	¹⁸ MPI_Comm_test_inter(comm, flag, ierror) BIND(C)
	¹⁹ TYPE(MPI_Comm), INTENT(IN) :: comm
	²⁰ LOGICAL, INTENT(OUT) :: flag
	²¹ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	
	MPI_Group_compare(group1, group2, result, ierror) BIND(C)
	<pre>24 TYPE(MPI_Group), INTENT(IN) :: group1, group2 25 INTEGER, INTENT(OUT) :: result</pre>
	INTEGED OPTIONAL INTENT(OUT) · · · ·
ticket- $248T$.	27
	MP1_Group_difference(group1, group2, newgroup, ierror) BIND(C)
	TYPE(MP1_Group), INTENT(IN) :: group1, group2
	IYPE(MPI_Group), INIENI(UUI) :: newgroup
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C)
	33 TYPE(MPI_Group), INTENT(IN) :: group
	INTEGER, INTENT(IN) :: n, ranks(n)
	35 TYPE(MPI_Group), INTENT(OUT) :: newgroup
ticket-248T.	³⁶ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	³⁷ MPI_Group_free(group, ierror) BIND(C)
	³⁸ TYPE(MPI_Group), INTENT(INOUT) :: group
ticket-248T	³⁹ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	41 MPI_Group_incl(group, n, ranks, newgroup, ierror) BIND(C)
	42 TYPE(MPI_Group), INTENT(IN) :: group
	43 INTEGER, INTENT(IN) :: n, ranks(n)
	44 TYPE(MPI_Group), INTENT(OUT) :: newgroup
ticket-248T	45 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	⁴⁶ MPI Group intersection (group1 group2 neugroup ierror) BIND(C)
	⁴⁷ MPI_Group_intersection(group1, group2, newgroup, ierror) BIND(C) TYPE(MPI_Group), INTENT(IN) :: group1, group2
	48

TYPE(MPI_Group), INTENT(OUT) :: newgroup	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$\frac{2}{1}$
	$_{3}$ ticket-248T.
MPI_Group_range_excl(group, n, ranges, newgroup, ierror) BIND(C)	4
TYPE(MPI_Group), INTENT(IN) :: group	5
<pre>INTEGER, INTENT(IN) :: n, ranges(3,n)</pre>	6
TYPE(MPI_Group), INTENT(OUT) :: newgroup	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8 ticket-248T.
MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)	9 UICKet-2401.
	10
TYPE(MPI_Group), INTENT(IN) :: group	11
<pre>INTEGER, INTENT(IN) :: n, ranges(3,n)</pre>	12
TYPE(MPI_Group), INTENT(OUT) :: newgroup	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$^{13}_{14}$ ticket-248T.
MPI_Group_rank(group, rank, ierror) BIND(C)	
TYPE(MPI_Group), INTENT(IN) :: group	15
INTEGER, INTENT(OUT) :: rank	16
	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18 ticket-248T.
MPI_Group_size(group, size, ierror) BIND(C)	19
TYPE(MPI_Group), INTENT(IN) :: group	20
INTEGER, INTENT(OUT) :: size	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	23 ticket-248T.
<pre>MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)</pre>	24
BIND(C)	25
TYPE(MPI_Group), INTENT(IN) :: group1, group2	26
INTEGER, INTENT(IN) :: n, ranks1(n)	27
INTEGER, INTENT(OUT) :: ranks2(n)	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	$^{29}_{30}$ ticket-248T.
MPI_Group_union(group1, group2, newgroup, ierror) BIND(C)	31
TYPE(MPI_Group), INTENT(IN) :: group1, group2	
TYPE(MPI_Group), INTENT(OUT) :: newgroup	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
NDT Intercomm excets(local comm local locator more comm remote locator	³⁴ ticket-248T.
<pre>MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,</pre>	35
tag, newintercomm, ierror) BIND(C)	36
TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm	37
<pre>INTEGER, INTENT(IN) :: local_leader, remote_leader, tag</pre>	38
TYPE(MPI_Comm), INTENT(OUT) :: newintercomm	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40 ticket-248T.
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) BIND(C)	41 UCKet-2401.
TYPE(MPI_Comm), INTENT(IN) :: intercomm	42
LOGICAL, INTENT(IN) :: high	43
-	44
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	⁴⁶ ticket-248T.
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,	47
extra_state, ierror) BIND(C)	48

	1	PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
	2	PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
	3	INTEGER, INTENT(OUT) :: type_keyval
	4	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
	5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	6	
	7	<pre>MPI_Type_delete_attr(datatype, type_keyval, ierror) BIND(C)</pre>
	8	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	9	INTEGER, INTENT(IN) :: type_keyval
ticket-248T.	10	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UCKC0-2401.	11	MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
	12	attribute_val_out, flag, ierror) BIND(C)
	13	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	14	INTEGER, INTENT(IN) :: type_keyval
	15	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
	16	attribute_val_in
	17	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
	18	LOGICAL, INTENT(OUT) :: flag
	19	INTEGER, INTENT(OUT) :: ierror
ticket-248T.	20	
	21	MPI_Type_free_keyval(type_keyval, ierror) BIND(C)
	22	INTEGER, INTENT(INOUT) :: type_keyval
ticket-248T.	23	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	24	MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
	25	BIND(C)
	26	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	27	<pre>INTEGER, INTENT(IN) :: type_keyval</pre>
	28	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
	29	LOGICAL, INTENT(OUT) :: flag
+:-l+ 040T	30	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.		MPI_Type_get_name(datatype, type_name, resultlen, ierror) BIND(C)
	32	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	33	CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
	34	INTEGER, INTENT(OUT) :: resultlen
	35	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	36	
	37	<pre>MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,</pre>
	38 39	<pre>attribute_val_out, flag, ierror) BIND(C)</pre>
	40	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
	40	<pre>INTEGER, INTENT(IN) :: type_keyval</pre>
	42	<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,</pre>
	43	attribute_val_in
	43	<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out</pre>
	44	LOGICAL, INTENT(OUT) :: flag
ticket-248T.		INTEGER, INTENT(OUT) :: ierror
JUNU 2101.	40	MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
	48	ierror) BIND(C)
	-	

TYPE(MPI_Datatype), INTENT(IN) :: datatype	1
INTEGER, INTENT(IN) :: type_keyval	2
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,	3
extra_state	4
INTEGER, INTENT(OUT) :: ierror	5
,,	$_{6}$ ticket-248T.
<pre>MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)</pre>	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
INTEGER, INTENT(IN) :: type_keyval	9
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
	$^{11}_{12}$ ticket-248T.
MPI_Type_set_name(datatype, type_name, ierror) BIND(C)	13
TYPE(MPI_Datatype), INTENT(IN) :: datatype	14
CHARACTER(LEN=*), INTENT(IN) :: type_name	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$^{15}_{16}$ ticket-248T.
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,	
extra_state, ierror) BIND(C)	17
PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn	18
PROCEDURE(MPI_WIN_COPY_attr_function) win_copy_attr_fn PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn	19
INTEGER, INTENT(OUT) :: win_keyval	20
•	21
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23 ticket-248T.
MPI_Win_delete_attr(win, win_keyval, ierror) BIND(C)	24
TYPE(MPI_Win), INTENT(IN) :: win	25
INTEGER, INTENT(IN) :: win_keyval	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	$_{28}$ ticket-248T.
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,	29
<pre>attribute_val_out, flag, ierror) BIND(C)</pre>	30
INTEGER, INTENT(IN) :: oldwin, win_keyval	31
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,</pre>	32
attribute_val_in	33
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out	34
LOGICAL, INTENT(OUT) :: flag	35
INTEGER, INTENT(OUT) :: ierror	36
	$_{37}^{30}$ ticket-248T.
MPI_Win_free_keyval(win_keyval, ierror) BIND(C)	38
INTEGER, INTENT(INOUT) :: win_keyval	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40 ticket-248T.
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror) BIND(C)	40 ticket-2401.
TYPE(MPI_Win), INTENT(IN) :: win	
INTEGER, INTENT(IN) :: win_keyval	42
INTEGER, INTENT(IN) WIILREYVAL INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val	43
	44
LOGICAL, INTENT(OUT) :: flag	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46 ticket-248T.
MPI_Win_get_name(win, win_name, resultlen, ierror) BIND(C)	47
	48

ticket-248T.	6 7 8 9	<pre>TYPE(MPI_Win), INTENT(IN) :: win CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C) INTEGER, INTENT(IN) :: oldwin, win_keyval INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state, attribute_val_in</pre>
ticket-248T.	10 11 12 13 14	<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out LOGICAL, INTENT(OUT) :: flag INTEGER, INTENT(OUT) :: ierror</pre>
	14 15 16 17 18 19	<pre>MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror) BIND(C) TYPE(MPI_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: win_keyval INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val, extra state</pre>
ticket-248T.	20 21	<pre>extra_state INTEGER, INTENT(OUT) :: ierror MPI_Win_set_attr(win, win_keyval, attribute_val, ierror) BIND(C)</pre>
ticket-248T.	22 23 24 25 26	<pre>TYPE(MPI_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: win_keyval INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
UICKET-2401.	27 28 29 30 31 32	<pre>MPI_Win_set_name(win, win_name, ierror) BIND(C) TYPE(MPI_Win), INTENT(IN) :: win CHARACTER(LEN=*), INTENT(IN) :: win_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
ticket-248T.	33 34 35 36	<pre>A.3.5 Process Topologies Fortran 2008 Bindings MPI_Cart_coords(comm, rank, maxdims, coords, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: rank, maxdims</pre>
ticket-248T.	37 38 39	INTEGER, INTENT(OUT) :: coords(maxdims) INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	40 41 42 43 44 45 46	<pre>MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm_old INTEGER, INTENT(IN) :: ndims, dims(ndims) LOGICAL, INTENT(IN) :: periods(ndims), reorder TYPE(MPI_Comm), INTENT(OUT) :: comm_cart INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
ticket-248T.	40 47 48	MPI_Cartdim_get(comm, ndims, ierror) BIND(C)

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   2
    INTEGER, INTENT(OUT) :: ndims
                                                                                   3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   _4 ticket-248T.
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) BIND(C)
                                                                                   5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   6
    INTEGER, INTENT(IN) :: maxdims
                                                                                   7
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
                                                                                   8
    LOGICAL, INTENT(OUT) :: periods(maxdims)
                                                                                   9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  ^{10} ticket-248T.
                                                                                  11
MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) BIND(C)
                                                                                  12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  13
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
                                                                                  14
    LOGICAL, INTENT(IN) :: periods(ndims)
                                                                                  15
    INTEGER, INTENT(OUT) :: newrank
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17 ticket-248T.
MPI_Cart_rank(comm, coords, rank, ierror) BIND(C)
                                                                                  18
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  19
    INTEGER, INTENT(IN) :: coords(*)
                                                                                  20
    INTEGER, INTENT(OUT) :: rank
                                                                                  21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  22
                                                                                    ticket-248T.
                                                                                  23
MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
                                                                                  ^{24}
             BIND(C)
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    INTEGER, INTENT(IN) :: direction, disp
                                                                                  27
    INTEGER, INTENT(OUT) :: rank_source, rank_dest
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  <sup>29</sup> ticket-248T.
MPI_Cart_sub(comm, remain_dims, newcomm, ierror) BIND(C)
                                                                                  30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  31
    LOGICAL, INTENT(IN) :: remain_dims(*)
                                                                                  32
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                    ticket-248T.
                                                                                  35
MPI_Dims_create(nnodes, ndims, dims, ierror) BIND(C)
                                                                                  36
    INTEGER, INTENT(IN) :: nnodes, ndims
                                                                                  37
    INTEGER, INTENT(INOUT) :: dims(ndims)
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  <sup>39</sup> ticket-248T.
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                                                                                  40
             outdegree, destinations, destweights, info, reorder,
                                                                                  41
              comm_dist_graph, ierror) BIND(C)
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  43
    INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
                                                                                  44
    destinations(outdegree)
                                                                                  45
    INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
                                                                                  46
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   47
    LOGICAL, INTENT(IN) :: reorder
                                                                                   48
```

```
1
                   TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
           2
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 3
               MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
           4
                             info, reorder, comm_dist_graph, ierror) BIND(C)
           5
                   TYPE(MPI_Comm), INTENT(IN) :: comm_old
           6
                   INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
           7
                   INTEGER, INTENT(IN) :: weights(*)
           8
                   TYPE(MPI_Info), INTENT(IN) :: info
           9
                   LOGICAL, INTENT(IN) :: reorder
          10
                   TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
          11
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. ^{12}
          13
               MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
          14
                             maxoutdegree, destinations, destweights, ierror) BIND(C)
          15
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          16
                   INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
          17
                   INTEGER, INTENT(OUT) :: sources(maxindegree),
          18
                   destinations(maxoutdegree)
          19
                   INTEGER :: sourceweights(*), destweights(*)
          20
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 21
               MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
          22
                             BIND(C)
          23
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          24
                   INTEGER, INTENT(OUT) :: indegree, outdegree
          25
                   LOGICAL, INTENT(OUT) :: weighted
          26
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          27
ticket-248T.
          28
               MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
          29
                             ierror) BIND(C)
          30
                   TYPE(MPI_Comm), INTENT(IN) :: comm_old
          31
                   INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
          32
                   LOGICAL, INTENT(IN) :: reorder
          33
                   TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
          34
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35
               MPI_Graphdims_get(comm, nnodes, nedges, ierror) BIND(C)
          36
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          37
                   INTEGER, INTENT(OUT) :: nnodes, nedges
          38
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          39
ticket-248T.
          40
               MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) BIND(C)
          41
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          42
                   INTEGER, INTENT(IN) :: maxindex, maxedges
          43
                   INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
          44
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 45
               MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) BIND(C)
          46
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          47
                   INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
          48
```

```
1
    INTEGER, INTENT(OUT) :: newrank
                                                                                  2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  _{3} ticket-248T.
MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) BIND(C)
                                                                                  4
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  5
    INTEGER, INTENT(IN) :: rank, maxneighbors
                                                                                  6
    INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
                                                                                  7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  8
                                                                                    ticket-248T.
                                                                                  a
MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) BIND(C)
                                                                                  10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  11
    INTEGER, INTENT(IN) :: rank
    INTEGER, INTENT(OUT) :: nneighbors
                                                                                  12
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14 ticket-248T.
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  15
             recvtype, comm, request, ierror) BIND(C)
                                                                                  16
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  17
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  18
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
                                                                                    ticket-248T.
                                                                                  24
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                  25
             displs, recvtype, comm, request, ierror) BIND(C)
                                                                                  26
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  28
    INTEGER, INTENT(IN) :: sendcount
                                                                                  29
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  <sup>34</sup> ticket-248T.
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  35
             recvtype, comm, request, ierror) BIND(C)
                                                                                  36
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  37
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  38
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  39
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  41
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
                                                                                    ticket-248T.
                                                                                  44
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  45
             recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)
                                                                                  46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  48
```

```
1
                   INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
           2
                   recvcounts(*), rdispls(*)
           3
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           4
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           5
                   TYPE(MPI_Request), INTENT(OUT) :: request
           6
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
               MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
           8
                             recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
           9
                   TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
           10
                   TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
           11
                   INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
          12
                    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
          13
                   sdispls(*), rdispls(*)
          14
                   TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
          15
                   recvtypes(*)
          16
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           17
                   TYPE(MPI_Request), INTENT(OUT) :: request
           18
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. ^{19}
          20
                MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
          21
                             recvtype, comm, ierror) BIND(C)
          22
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          23
                   TYPE(*), DIMENSION(..) :: recvbuf
          24
                   INTEGER, INTENT(IN) :: sendcount, recvcount
           25
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           26
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          27
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 28
               MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
          29
                             displs, recvtype, comm, ierror) BIND(C)
          30
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          31
                   TYPE(*), DIMENSION(..) :: recvbuf
          32
                   INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
          33
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
          34
                   TYPE(MPI_Comm), INTENT(IN) :: comm
          35
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          36
ticket-248T.
          37
                MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
          38
                             recvtype, comm, ierror) BIND(C)
          39
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          40
                   TYPE(*), DIMENSION(...) :: recvbuf
          41
                   INTEGER, INTENT(IN) :: sendcount, recvcount
          42
                   TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
           43
                   TYPE(MPI_Comm), INTENT(IN) :: comm
           44
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 45
               MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
           46
                             recvcounts, rdispls, recvtype, comm, ierror) BIND(C)
           47
                   TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
           48
```

```
TYPE(*), DIMENSION(..) :: recvbuf
                                                                                    1
                                                                                    \mathbf{2}
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
                                                                                    4
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    5
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    6
                                                                                    7 ticket-248T.
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                    8
              recvcounts, rdispls, recvtypes, comm, ierror) BIND(C)
                                                                                    9
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                    10
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                    11
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                    12
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                    13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                    14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    ^{16} ticket-248T.
                                                                                    17
MPI_Topo_test(comm, status, ierror) BIND(C)
                                                                                    18
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    19
    INTEGER, INTENT(OUT) :: status
                                                                                    20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    21
                                                                                    22
A.3.6 MPI Environmental Management Fortran 2008 Bindings
                                                                                    23
                                                                                    _{24} ticket-248T.
DOUBLE PRECISION MPI_Wtick() BIND(C)
                                                                                    <sup>25</sup> ticket-248T.
DOUBLE PRECISION MPI_Wtime() BIND(C)
                                                                                    26
                                                                                    _{27} ticket-248T.
MPI_Abort(comm, errorcode, ierror) BIND(C)
                                                                                    28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    29
    INTEGER, INTENT(IN) :: errorcode
                                                                                    30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    ^{31} ticket-248T.
MPI_Add_error_class(errorclass, ierror) BIND(C)
                                                                                    32
    INTEGER, INTENT(OUT) :: errorclass
                                                                                    33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    34
                                                                                    _{35} ticket-248T.
MPI_Add_error_code(errorclass, errorcode, ierror) BIND(C)
                                                                                    36
    INTEGER, INTENT(IN) :: errorclass
                                                                                    37
    INTEGER, INTENT(OUT) :: errorcode
                                                                                    38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    ^{\rm 39} ticket-248T.
MPI_Add_error_string(errorcode, string, ierror) BIND(C)
                                                                                    40
    INTEGER, INTENT(IN) :: errorcode
                                                                                    41
    CHARACTER(LEN=*), INTENT(IN) :: string
                                                                                    42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    43
                                                                                    44 ticket-248T.
MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
                                                                                    45
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    46
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                    47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    48
```

	1	TYPE(C_PTR), INTENT(OUT) :: baseptr
	2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T	• 3	MDI Commencial contraction in the DIND (C)
	4	MPI_Comm_call_errhandler(comm, errorcode, ierror) BIND(C)
	5	TYPE(MPI_Comm), INTENT(IN) :: comm
	6	INTEGER, INTENT(IN) :: errorcode
ticket-248T	7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	8	MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) BIND(C)
	9	PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
	10	TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
	11	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T	• 12	
	13	MPI_Comm_get_errhandler(comm, errhandler, ierror) BIND(C)
	14	TYPE(MPI_Comm), INTENT(IN) :: comm
	15	TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
ticket-248T	16	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	17	MPI_Comm_set_errhandler(comm, errhandler, ierror) BIND(C)
	18	TYPE(MPI_Comm), INTENT(IN) :: comm
	19	TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
	20	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T	. 21	
	22	MPI_Errhandler_free(errhandler, ierror) BIND(C)
	23	TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
ticket-248T	24	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
01CKC0 2401	25	MPI_Error_class(errorcode, errorclass, ierror) BIND(C)
	26	INTEGER, INTENT(IN) :: errorcode
	27	INTEGER, INTENT(OUT) :: errorclass
	28	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T		
	30	MPI_Error_string(errorcode, string, resultlen, ierror) BIND(C)
	31	INTEGER, INTENT(IN) :: errorcode
	32	CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
	33	INTEGER, INTENT(OUT) :: resultlen
ticket-248T	34	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
01CIKC0 2401	35	MPI_File_call_errhandler(fh, errorcode, ierror) BIND(C)
	36	TYPE(MPI_File), INTENT(IN) :: fh
	37	INTEGER, INTENT(IN) :: errorcode
	38	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T	. 39	
	40	<pre>MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C)</pre>
	41	<pre>PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn</pre>
	42	TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
ticket-248T	43	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UICKEU-2401	• 44	MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)
	45	TYPE(MPI_File), INTENT(IN) :: file
	46	TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
	47	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T	. 48	

MPI_File_set_errhandler(file, errhandler, ierror) BIND(C)	1
TYPE(MPI_File), INTENT(IN) :: file	2
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	$_5$ ticket-248T.
MPI_Finalized(flag, ierror) BIND(C)	6
LOGICAL, INTENT(OUT) :: flag	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8 ticket-248T.
MPI_Finalize(ierror) BIND(C)	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	$_{11}$ ticket-248T.
MPI_Free_mem(base, ierror) BIND(C)	12
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: base	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	¹⁴ , 1, 0, 10TD
NDI (at an accord and (none accultion former) DIND(C)	$^{14}_{15}$ ticket-248T.
MPI_Get_processor_name(name, resultlen, ierror) BIND(C)	16
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name	17
INTEGER, INTENT(OUT) :: resultlen	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	¹⁹ ticket-248T.
MPI_Get_version(version, subversion, ierror) BIND(C)	20
INTEGER, INTENT(OUT) :: version, subversion	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	$^{22}_{23}$ ticket-248T.
MPI_Initialized(flag, ierror) BIND(C)	23
LOGICAL, INTENT(OUT) :: flag	25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25 26 ticket-248T.
MPI_Init(ierror) BIND(C)	20 UICKEU-240 I.
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
	$\frac{28}{10}$ ticket-248T.
<pre>MPI_Win_call_errhandler(win, errorcode, ierror) BIND(C)</pre>	29
TYPE(MPI_Win), INTENT(IN) :: win	30
INTEGER, INTENT(IN) :: errorcode	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
NDT Uin weste ender Der (win ender Der fre ender Der iennen) DIND(()	33 ticket-248T.
MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror) BIND(C)	34
PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn	35
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$^{37}_{_{38}}$ ticket-248T.
MPI_Win_get_errhandler(win, errhandler, ierror) BIND(C)	38
TYPE(MPI_Win), INTENT(IN) :: win	39
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	42 ticket-248T.
MPI_Win_set_errhandler(win, errhandler, ierror) BIND(C)	43
TYPE(MPI_Win), INTENT(IN) :: win	44
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46
	47
	48

1 The Info Object Fortran 2008 Bindings A.3.7 ticket-248T. 2 MPI_Info_create(info, ierror) BIND(C) 3 TYPE(MPI_Info), INTENT(OUT) :: info 4 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5ticket-248T. 6 MPI_Info_delete(info, key, ierror) BIND(C) 7 TYPE(MPI_Info), INTENT(IN) :: info 8 CHARACTER(LEN=*), INTENT(IN) :: key 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 10 MPI_Info_dup(info, newinfo, ierror) BIND(C) 11 TYPE(MPI_Info), INTENT(IN) :: info 12TYPE(MPI_Info), INTENT(OUT) :: newinfo 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 14 15MPI_Info_free(info, ierror) BIND(C) 16TYPE(MPI_Info), INTENT(INOUT) :: info 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 18 MPI_Info_get(info, key, valuelen, value, flag, ierror) BIND(C) 19 TYPE(MPI_Info), INTENT(IN) :: info 20CHARACTER(LEN=*), INTENT(IN) :: key 21INTEGER, INTENT(IN) :: valuelen 22 CHARACTER(LEN=valuelen), INTENT(OUT) :: value 23 LOGICAL, INTENT(OUT) :: flag 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25ticket-248T. 26MPI_Info_get_nkeys(info, nkeys, ierror) BIND(C) 27TYPE(MPI_Info), INTENT(IN) :: info 28INTEGER, INTENT(OUT) :: nkeys 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 30 MPI_Info_get_nthkey(info, n, key, ierror) BIND(C) 31 TYPE(MPI_Info), INTENT(IN) :: info 32 INTEGER, INTENT(IN) :: n 33 CHARACTER(LEN=*), INTENT(OUT) :: key 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35ticket-248T. 36 MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) BIND(C) 37 TYPE(MPI_Info), INTENT(IN) :: info 38 CHARACTER(LEN=*), INTENT(IN) :: key 39 INTEGER, INTENT(OUT) :: valuelen 40 LOGICAL, INTENT(OUT) :: flag 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror ticket-248T. 42 MPI_Info_set(info, key, value, ierror) BIND(C) 43 TYPE(MPI_Info), INTENT(IN) :: info 44 CHARACTER(LEN=*), INTENT(IN) :: key, value 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46 47 48

A.S. FORTRAN 2008 DINDINGS WITH THE MIT_F08 MODULE	09
A.3.8 Process Creation and Management Fortran 2008 Bindings	1
	$_2$ ticket-248T.
MPI_Close_port(port_name, ierror) BIND(C)	3
CHARACTER(LEN=*), INTENT(IN) :: port_name	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5 ticket-248T.
MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) BIND(C)	6 6 6 CICKED-2401.
CHARACTER(LEN=*), INTENT(IN) :: port_name	7
TYPE(MPI_Info), INTENT(IN) :: info	8
INTEGER, INTENT(IN) :: root	9
TYPE(MPI_Comm), INTENT(IN) :: comm	10
TYPE(MPI_Comm), INTENT(UN) :: newcomm	11
	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$_{13}$ ticket-248T.
MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror) BIND(C)	14
CHARACTER(LEN=*), INTENT(IN) :: port_name	15
TYPE(MPI_Info), INTENT(IN) :: info	16
INTEGER, INTENT(IN) :: root	17
TYPE(MPI_Comm), INTENT(IN) :: comm	18
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	$^{20}_{21}$ ticket-248T.
MPI_Comm_disconnect(comm, ierror) BIND(C)	22
TYPE(MPI_Comm), INTENT(INOUT) :: comm	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	²⁰ ₂₄ ticket-248T.
MPI_Comm_get_parent(parent, ierror) BIND(C)	25
TYPE(MPI_Comm), INTENT(OUT) :: parent	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	$^{27}_{28}$ ticket-248T.
MPI_Comm_join(fd, intercomm, ierror) BIND(C)	29
INTEGER, INTENT(IN) :: fd	30
TYPE(MPI_Comm), INTENT(OUT) :: intercomm	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$_{32}$ ticket-248T.
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,	33
array_of_errcodes, ierror) BIND(C)	34
CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)	35
INTEGER, INTENT(IN) :: maxprocs, root	36
TYPE(MPI_Info), INTENT(IN) :: info	37
TYPE(MPI_Comm), INTENT(IN) :: comm	38
TYPE(MPI_Comm), INTENT(OUT) :: intercomm	39
INTEGER :: array_of_errcodes(*)	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	$_{42}^{41}$ ticket-248T.
MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,	43
<pre>array_of_maxprocs, array_of_info, root, comm, intercomm,</pre>	43
array_of_errcodes, ierror) BIND(C)	45
<pre>INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root</pre>	46
CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),	40
<pre>array_of_argv(count, *)</pre>	48

789

A.3. FORTRAN 2008 BINDINGS WITH THE MPI_F08 MODULE

	1	
		<pre>TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)</pre>
	2	TYPE(MPI_Comm), INTENT(IN) :: comm
	3	TYPE(MPI_Comm), INTENT(OUT) :: intercomm
	4	<pre>INTEGER :: array_of_errcodes(*)</pre>
1.1 1 040m	5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	•	MDT Lookun nome (counties nome info nont nome issues) DIND(()
	7	MPI_Lookup_name(service_name, info, port_name, ierror) BIND(C)
	8	CHARACTER(LEN=*), INTENT(IN) :: service_name
	9	TYPE(MPI_Info), INTENT(IN) :: info
	10	CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
ticket-248T.	11	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
01CKC0-2401.		MPI_Open_port(info, port_name, ierror) BIND(C)
	13	TYPE(MPI_Info), INTENT(IN) :: info
	14	CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
	15	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	16	INIDODA, OFFICERE, INTENT(U01/ TETTOL
		<pre>MPI_Publish_name(service_name, info, port_name, ierror) BIND(C)</pre>
	18	TYPE(MPI_Info), INTENT(IN) :: info
	19	CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
	20	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.		
	21	<pre>MPI_Unpublish_name(service_name, info, port_name, ierror) BIND(C)</pre>
		CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
	23	TYPE(MPI_Info), INTENT(IN) :: info
	24	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	25	
	26	A 2.0 One Sided Communications Fortune 2008 Diadians
ticket-248T.	27	A.3.9 One-Sided Communications Fortran 2008 Bindings
		MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
	29	<pre>target_disp, target_count, target_datatype, op, win, ierror)</pre>
	30	
		BIND(C)
	31	BIND(C)
	32	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr
		BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
	32	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
	32 33	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
	32 33 34	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op
	32 33 34 35	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Op), INTENT(IN) :: win
ticket-248T.	32 33 34 35 36 37	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op
ticket-248T.	32 33 34 35 36 37 38	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Op), INTENT(IN) :: win
ticket-248T.	32 33 34 35 36 37 38	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Uin), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	32 33 34 35 36 37 38 39	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
ticket-248T.	32 33 34 35 36 37 38 39 40	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Op), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C)
ticket-248T.	32 33 34 35 36 37 38 39 40 41	<pre>BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr,</pre>
ticket-248T.	32 33 34 35 36 37 38 39 40 41 42	<pre>BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Op), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr, compare_addr</pre>
ticket-248T.	32 33 34 35 36 37 38 39 40 41 42 43	<pre>BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Op), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr, compare_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr</pre>
ticket-248T.	32 33 34 35 36 37 38 39 40 41 42 43 44	<pre>BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr, compare_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr TYPE(*), DIMENSION(), INTENT(IN) :: datatype</pre>
ticket-248T.	32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr, compare_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr TYPE(*), DIMENSION(), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: target_rank</pre>
ticket-248T.	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<pre>BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr, compare_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr TYPE(*), DIMENSION(), INTENT(IN) :: target_rank INTEGER, INTENT(IN) :: target_rank INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp</pre>

```
1
                                                                                  _2 ticket-248T.
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
             target_disp, op, win, ierror) BIND(C)
                                                                                  3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  4
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                  5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  6
    INTEGER, INTENT(IN) :: target_rank
                                                                                  7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  8
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  9
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
                                                                                  _{12} ticket-248T.
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
                                                                                  13
             result_count, result_datatype, target_rank, target_disp,
                                                                                  14
             target_count, target_datatype, op, win, ierror) BIND(C)
                                                                                  15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  16
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
                                                                                  17
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                  18
    target_count
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
                                                                                  20
    result_datatype
                                                                                  21
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  22
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  23
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  ^{24}
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                        ierror
                                                                                  ^{25} ticket-248T.
                                                                                  26
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
                                                                                  27
             BIND(C)
                                                                                  28
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
                                                                                  29
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  31
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  32
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  <sub>35</sub> ticket-248T.
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  36
             target_disp, target_count, target_datatype, win, ierror)
                                                                                  37
             BIND(C)
                                                                                  38
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  39
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  41
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  42
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  ^{44} ticket-248T.
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  45
             target_disp, target_count, target_datatype, op, win, request,
                                                                                  46
                                                                                  47
             ierror) BIND(C)
                                                                                  48
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
```

	1	INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
	2	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
	3	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
	4	TYPE(MPI_Op), INTENT(IN) :: op
	5	TYPE(MPI_Win), INTENT(IN) :: win
	6	TYPE(MPI_Request), INTENT(OUT) :: request
	7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	8	
	9	<pre>MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,</pre>
	10	result_addr, result_count, result_datatype, target_rank,
	11	<pre>target_disp, target_count, target_datatype, op, win, request,</pre>
	12	ierror) BIND(C)
	13	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>
	14	TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr
	15	<pre>INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,</pre>
	16	target_count
		TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
	17	result_datatype
	18	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
	19	TYPE(MPI_Op), INTENT(IN) :: op
	20	TYPE(MPI_Win), INTENT(IN) :: win
	21	TYPE(MPI_Request), INTENT(OUT) :: request
	22	
ticket-248T.	23	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	24	MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
	25	<pre>target_disp, target_count, target_datatype, win, request,</pre>
	26	ierror) BIND(C)
	27	TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr
	28	INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
	29	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
	30	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
	31	TYPE(MPI_Win), INTENT(IN) :: win
	32	TYPE(MPI_Request), INTENT(OUT) :: request
	33	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	34	
	35	MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
	36	<pre>target_disp, target_count, target_datatype, win, request,</pre>
	37	ierror) BIND(C)
	38	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr
	39	INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
	40	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
	41	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
		TYPE(MPI_Win), INTENT(IN) :: win
	42	TYPE(MPI_Request), INTENT(OUT) :: request
	43	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	44	INIDALA, DITIONAL, INIDAL(UUI/ IGIIOI
	45	<pre>MPI_Win_allocate_shared(size, info, comm, baseptr, win, ierror) BIND(C)</pre>
	46	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
	47	<pre>INTEGER(KIND=MPI_Address_kind), INTENT(IN) :: size</pre>
	48	

TYPE(MPI_Info), INTENT(IN) :: info	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2
TYPE(C_PTR), INTENT(OUT) :: baseptr	3
TYPE(MPI_Win), INTENT(OUT) :: win	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
	$_{6}$ ticket-248T.
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) BIND(C)	7
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	8
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size	9
INTEGER, INTENT(IN) :: disp_unit	10
TYPE(MPI_Info), INTENT(IN) :: info	11
TYPE(MPI_Comm), INTENT(IN) :: comm	12
TYPE(C_PTR), INTENT(OUT) :: baseptr	13
TYPE(MPI_Win), INTENT(OUT) :: win	14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15 ticket-248T.
MPI_Win_attach(win, base, size, ierror) BIND(C)	16
TYPE(MPI_Win), INTENT(IN) :: win	17
TYPE(*), ASYNCHRONOUS :: base	18
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	$_{21}$ ticket-248T.
MPI_Win_complete(win, ierror) BIND(C)	22
TYPE(MPI_Win), INTENT(IN) :: win	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24 ticket-248T.
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) BIND(C)	25
TYPE(*), DIMENSION(), ASYNCHRONOUS :: base	26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size	27
INTEGER, INTENT(IN) :: disp_unit	28
TYPE(MPI_Info), INTENT(IN) :: info	29
TYPE(MPI_Comm), INTENT(IN) :: comm	30
TYPE(MPI_Win), INTENT(OUT) :: win	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
	33 ticket-248T.
MPI_Win_create_dynamic(info, comm, win, ierror) BIND(C)	34
TYPE(MPI_Info), INTENT(IN) :: info	35
TYPE(MPI_Comm), INTENT(IN) :: comm	36
TYPE(MPI_Win), INTENT(OUT) :: win	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38 ticket-248T.
MPI_Win_detach(win, base, ierror) BIND(C)	39
TYPE(MPI_Win), INTENT(IN) :: win	40
TYPE(*), ASYNCHRONOUS :: base	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MDT Uin for as (accent win isoman) DIND(2)	⁴³ ticket-248T.
MPI_Win_fence(assert, win, ierror) BIND(C)	44
INTEGER, INTENT(IN) :: assert	45
TYPE(MPI_Win), INTENT(IN) :: win	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
	47 ticket-248T.

	¹ MPI Win fluch all (win increar) $RIND(C)$
	MI_WIN_IIUSN_AII(WIN, IEIIOI) DIND(0)
	IIFE(MFI_WIII), INTENI(IN) WIII
ticket-248T.	INTEGER, UFITUNAL, INTENT(UUT) IETTUT
	5 MPI_Win_flush_local_all(win, ierror) BIND(C)
	G TYPE(MPI_Win), INTENT(IN) :: win
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	
	<pre>8 MPI_Win_flush_local(rank, win, ierror) BIND(C) 9 INTEGED_INTENT(IN) + month</pre>
	INTEGER, INTENI(IN) :: IAIIK
	IIPE(MPI_WIN), INTENI(IN) :: WIN
ticket-248T.	INTEGER, UPITUMAL, INTENI(UUI) :: IETTOT
0101100 21011	MPI_Win_flush(rank, win, ierror) BIND(C)
	INTEGER, INTENT(IN) :: rank
	TYPE(MPI_Win), INTENT(IN) :: win
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	
	MP1_win_iree(win, ierror) BIND(C)
	IYPE(MPI_win), INIENI(INUUI) :: Win
ticket-248T.	INIEGER, UPIIUNAL, INIENI(UUI) :: lerror
	MPI_Win_get_group(win, group, ierror) BIND(C)
	TYPE(MPI_Win), INTENT(IN) :: win
	TYPE(MPI_Group), INTENT(OUT) :: group
	124 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	95
	²³ MPI_Win_lock_all(assert, win, ierror) BIND(C)
	²⁰ INTEGER, INTENT(IN) :: assert ²⁷ TYPE (MDI Uin) INTENT(IN) as a raise
	²¹ TYPE(MPI_Win), INTENT(IN) :: win ²⁸ INTEGER OPTIONAL INTENT(OUT) is immer
ticket-248T.	²⁸ INTEGER, OPTIONAL, INTENT(OUT) :: ierror ²⁹
	MPI_Win_lock(lock_type, rank, assert, win, ierror) BIND(C)
	31 INTEGER, INTENT(IN) :: lock_type, rank, assert
	32 TYPE(MPI_Win), INTENT(IN) :: win
	33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	³⁴ MPI_Win_post(group, assert, win, ierror) BIND(C)
	³⁵ TYPE(MPI_Group), INTENT(IN) :: group
	³⁶ INTEGER, INTENT(IN) :: assert
	³⁷ TYPE(MPI_Win), INTENT(IN) :: win
	³⁸ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	³⁹
	40 MPI_Win_shared_query(win, rank, size, baseptr, ierror) BIND(C)
	41 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
	42 TYPE(MPI_Win), INTENT(IN) :: win
	43 INTEGER, INTENT(IN) :: rank
	44 INTEGER(KIND=MPI_Address_kind), INTENT(IN) :: size
	45 TYPE(C_PTR), INTENT(OUT) :: baseptr
ticket-248T.	46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UICKEU-2401.	⁴⁷ MPI_Win_start(group, assert, win, ierror) BIND(C)
	48 11 1_111_00010(group; abboro; 111, 101101; Dinb(0)

TYPE(MPI_Group), INTENT(IN) :: group	1
INTEGER, INTENT(IN) :: assert	2
TYPE(MPI_Win), INTENT(IN) :: win	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	$_5$ ticket-248T.
MPI_Win_sync(win, ierror) BIND(C)	6
TYPE(MPI_Win), INTENT(IN) :: win	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8 ticket-248T.
MPI_Win_test(win, flag, ierror) BIND(C)	9
TYPE(MPI_Win), INTENT(IN) :: win	10
LOGICAL, INTENT(OUT) :: flag	11
	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$_{13}$ ticket-248T.
MPI_Win_unlock_all(win, ierror) BIND(C)	14
TYPE(MPI_Win), INTENT(IN) :: win	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	$^{16}_{17}$ ticket-248T.
MPI_Win_unlock(rank, win, ierror) BIND(C)	-
INTEGER, INTENT(IN) :: rank	18
TYPE(MPI_Win), INTENT(IN) :: win	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_Win_wait(win, ierror) BIND(C)	$_{21}$ ticket-248T.
TYPE(MPI_Win), INTENT(IN) :: win	22
	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	25
A.3.10 External Interfaces Fortran 2008 Bindings	25 26
A.3.10 External Interfaces Fortran 2008 Bindings	26 ²⁷ ticket-248T.
MPI_Grequest_complete(request, ierror) BIND(C)	²⁶ ²⁷ ticket-248T. ²⁸
MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request	26 ²⁷ ticket-248T.
MPI_Grequest_complete(request, ierror) BIND(C)	26 27 ticket-248T. 28 29 30
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	26 27 ticket-248T. 28 29
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 29 30
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	 ²⁶ ²⁷ ticket-248T. ²⁸ ²⁹ ³⁰ ³¹ ticket-248T.
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 29 30 31 ticket-248T. 32
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	 ²⁶ ²⁷ ticket-248T. ²⁹ ³⁰ ³¹ ticket-248T. ³² ³³
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	 26 27 ticket-248T. 29 30 31 ticket-248T. 32 33 34
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35 36
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	 26 27 ticket-248T. 29 30 31 ticket-248T. 32 33 34 35 36 37
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35 36 37 38
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	 26 27 ticket-248T. 29 30 31 ticket-248T. 32 33 34 35 36 37 38 ³⁹ ticket-248T.
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror) BIND(C) PROCEDURE(MPI_Grequest_query_function) :: query_fn PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Init_thread(required, provided, ierror) BIND(C) INTEGER, INTENT(IN) :: required</pre>	 26 27 ticket-248T. 29 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40 41 42 43
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40 41 42
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror) BIND(C) PROCEDURE(MPI_Grequest_query_function) :: query_fn PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Init_thread(required, provided, ierror) BIND(C) INTEGER, INTENT(IN) :: required INTEGER, INTENT(OUT) :: provided INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Is_thread_main(flag, ierror) BIND(C)</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40 41 42 43
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror) BIND(C) PROCEDURE(MPI_Grequest_query_function) :: query_fn PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Init_thread(required, provided, ierror) BIND(C) INTEGER, INTENT(IN) :: required INTEGER, INTENT(OUT) :: ierror MPI_Is_thread_main(flag, ierror) BIND(C) LOGICAL, INTENT(OUT) :: flag</pre>	26 27 ticket-248T. 28 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40 41 42 43 44 ticket-248T.
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror) BIND(C) PROCEDURE(MPI_Grequest_query_function) :: query_fn PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Init_thread(required, provided, ierror) BIND(C) INTEGER, INTENT(IN) :: required INTEGER, INTENT(OUT) :: provided INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Is_thread_main(flag, ierror) BIND(C)</pre>	26 27 ticket-248T. 28 29 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40 41 42 43 44 ticket-248T. 45 46
<pre>MPI_Grequest_complete(request, ierror) BIND(C) TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror) BIND(C) PROCEDURE(MPI_Grequest_query_function) :: query_fn PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Init_thread(required, provided, ierror) BIND(C) INTEGER, INTENT(IN) :: required INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Is_thread_main(flag, ierror) BIND(C) LOGICAL, INTENT(OUT) :: flag</pre>	 26 27 ticket-248T. 29 30 31 ticket-248T. 32 33 34 35 36 37 38 39 ticket-248T. 40 41 42 43 44 ticket-248T. 45

	¹ INTEGER, INTENT(OUT) :: provided
ticket-248T.	² INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	MPI Status set cancelled(status flag jerror) BIND(C)
	TYPE(MPI_Status), INTENT(INOUT) :: status
	6 LOGICAL, INTENT(OUT) :: flag
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
UCKet-2401.	⁸ MPI_Status_set_elements(status, datatype, count, ierror) BIND(C)
	⁹ TYPE(MPI_Status), INTENT(INOUT) :: status
:	¹⁰ TYPE(MPI_Datatype), INTENT(IN) :: datatype
:	INTEGER, INTENT(IN) :: count
	¹² INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	
	MPI_Status_set_elements_x(status, datatype, count, ierror) BIND(C) TYPE(MPI_Status), INTENT(INOUT) :: status
	TYPE (MPI Datatura) INTENT (IN) · · datatura
	INTECEP (KIND - MDI COUNT KIND) INTENT(IN) ··· count
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	19
1	20 A 2.11 I/O Factors 2000 Dis lines
ticket-248T. ²	A.3.11 I/O Fortran 2008 Bindings
	MPI_File_close(fh, ierror) BIND(C)
1	TYPE(MPI_File), INTENT(INOUT) :: fh
ticket_248T	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
1	²⁵ MPI_File_delete(filename, info, ierror) BIND(C)
	CHARACTER(LEN=*), INTENT(IN) :: filename
	TYPE(MPI_Info), INTENT(IN) :: info
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	MPI_File_get_amode(fh, amode, ierror) BIND(C)
:	TYPE(MPI_File), INTENT(IN) :: fh
:	
	INTEGER, INTENT(OUT) :: amode
ticket 940T	TNTEGER OPTIONAL INTENT(OUT) ·· jerror
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI File). INTENT(IN) :: fh
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag
ticket-248T.	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror INTEGER, OPTIONAL, INTENT(OUT) :: ierror INTEGER, OPTIONAL, INTENT(IN) :: ierror INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C)
ticket-248T.	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh</pre>
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh
ticket-248T.	INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
ticket-248T.	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_group(fh, group, ierror) BIND(C)</pre>
ticket-248T.	<pre>MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_group(fh, group, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: ierror</pre>
ticket-248T.	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_atomicity(fh, flag, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_File_get_group(fh, group, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh </pre>

	1
NET File act info (fb info act) is an a DINE (g)	$_{2}$ ticket-248T.
MPI_File_get_info(fh, info_used, ierror) BIND(C)	2
TYPE(MPI_File), INTENT(IN) :: fh	3
TYPE(MPI_Info), INTENT(OUT) :: info_used	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
INTEGER, OFFICERED, INTERICOTY ICHICI	$_{6}$ ticket-248T.
MPI_File_get_position(fh, offset, ierror) BIND(C)	6
TYPE(MPI_File), INTENT(IN) :: fh	7
	8
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10 1 0.4000
	10 ticket-248T.
<pre>MPI_File_get_position_shared(fh, offset, ierror) BIND(C)</pre>	11
TYPE(MPI_File), INTENT(IN) :: fh	12
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
INTEGER, OFFICIARE, INTENT(DOT) TETTOT	$_{15}$ ticket-248T.
MPI_File_get_size(fh, size, ierror) BIND(C)	10
TYPE(MPI_File), INTENT(IN) :: fh	16
	17
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19 ticket-248T.
	ticket-2481.
<pre>MPI_File_get_type_extent(fh, datatype, extent, ierror) BIND(C)</pre>	
TYPE(MPI_File), INTENT(IN) :: fh	21
TYPE(MPI_Datatype), INTENT(IN) :: datatype	22
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	$_{25}$ ticket-248T.
<pre>MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) BIND(C)</pre>	26
TYPE(MPI_File), INTENT(IN) :: fh	
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp	27
•	28
TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype	29
CHARACTER(LEN=*), INTENT(OUT) :: datarep	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31 + 1 + 0.40
	$^{31}_{32}$ ticket-248T.
<pre>MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)</pre>	
BIND(C)	33
TYPE(MPI_File), INTENT(IN) :: fh	34
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	35
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	36
INTEGER, INTENT(IN) :: count	37
	38
TYPE(MPI_Datatype), INTENT(IN) :: datatype	39
TYPE(MPI_Request), INTENT(OUT) :: request	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	$_{41}$ ticket-248T.
<pre>MPI_File_iread(fh, buf, count, datatype, request, ierror) BIND(C)</pre>	42
TYPE(MPI_File), INTENT(IN) :: fh	43
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	44
INTEGER, INTENT(IN) :: count	45
TYPE(MPI_Datatype), INTENT(IN) :: datatype	40
III 2 (III 2 Data de gros), INIDAI (IN) data de gropo	10
TVDE(MDI Dogwogt) INTENT(OUT) ··· request	46
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46 47

	1	
ticket- $248T$.		MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) BIND(C)
	3	TYPE(MPI_File), INTENT(IN) :: fh
	4	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf
	5	INTEGER, INTENT(IN) :: count
	6	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	7	TYPE(MPI_Request), INTENT(OUT) :: request
	8	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.		
	10	<pre>MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)</pre>
	11	BIND(C)
	12	TYPE(MPI_File), INTENT(IN) :: fh
	13	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
	14	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	15	INTEGER, INTENT(IN) :: count
	16	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	17	TYPE(MPI_Request), INTENT(OUT) :: request
ticket-248T.	18	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	19	<pre>MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)</pre>
	20	TYPE(MPI_File), INTENT(IN) :: fh
	21	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	22	INTEGER, INTENT(IN) :: count
	23	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	24	TYPE(MPI_Request), INTENT(OUT) :: request
tialeat 949T	25	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.		MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
	27	TYPE(MPI_File), INTENT(IN) :: fh
	28	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	29	INTEGER, INTENT(IN) :: count
	30	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	31	TYPE(MPI_Request), INTENT(OUT) :: request
	32	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.		
	35	MPI_File_open(comm, filename, amode, info, fh, ierror) BIND(C)
	36	TYPE(MPI_Comm), INTENT(IN) :: comm
	37	CHARACTER(LEN=*), INTENT(IN) :: filename
	38	INTEGER, INTENT(IN) :: amode TYPE(MPI_Info), INTENT(IN) :: info
	39	TYPE(MPI_INIO), INTENT(IN) INTO TYPE(MPI_File), INTENT(OUT) :: fh
	40	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	41	INTEGER, OFITONAL, INTENT(001) TEITOT
	42	<pre>MPI_File_preallocate(fh, size, ierror) BIND(C)</pre>
	43	TYPE(MPI_File), INTENT(IN) :: fh
	44	<pre>INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size</pre>
ticket-248T.	45	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ucket-2401.		MPI_File_read_all_begin(fh, buf, count, datatype, ierror) BIND(C)
	47	TYPE(MPI_File), INTENT(IN) :: fh
	48	

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
                                                                                  1
                                                                                  ^{2}
    INTEGER, INTENT(IN) :: count
                                                                                  3
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
                                                                                  _5 ticket-248T.
MPI_File_read_all_end(fh, buf, status, ierror) BIND(C)
                                                                                  6
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  7
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
                                                                                  8
    TYPE(MPI_Status) :: status
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 ^{10} ticket-248T.
                                                                                  11
MPI_File_read_all(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                  12
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  13
    TYPE(*), DIMENSION(..) :: buf
                                                                                  14
    INTEGER, INTENT(IN) :: count
                                                                                  15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  16
    TYPE(MPI_Status) :: status
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18 ticket-248T.
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
                                                                                  19
             BIND(C)
                                                                                  20
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  21
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  22
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  23
    INTEGER, INTENT(IN) :: count
                                                                                  24
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 ^{26} ticket-248T.
                                                                                  27
MPI_File_read_at_all_end(fh, buf, status, ierror) BIND(C)
                                                                                  28
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  29
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
                                                                                  30
    TYPE(MPI_Status) :: status
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  <sup>32</sup> ticket-248T.
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
                                                                                  33
             BIND(C)
                                                                                  34
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  35
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  36
    TYPE(*), DIMENSION(..) :: buf
                                                                                  37
    INTEGER, INTENT(IN) :: count
                                                                                  38
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  39
    TYPE(MPI_Status) :: status
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    ticket-248T.
                                                                                  42
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
                                                                                  43
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  44
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  45
    TYPE(*), DIMENSION(..) :: buf
                                                                                  46
    INTEGER, INTENT(IN) :: count
                                                                                  47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  48
```

	¹ TYPE(MPI_Status) :: status
	² INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	•
	4 MPI_File_read(fh, buf, count, datatype, status, ierror) BIND(C)
	5 TYPE(MPI_File), INTENT(IN) :: fh TYPE(t) DIMENSION()
	6 TYPE(*), DIMENSION() :: buf
	7 INTEGER, INTENT(IN) :: count
	8 TYPE(MPI_Datatype), INTENT(IN) :: datatype
	9 TYPE(MPI_Status) :: status
ticket-248T.	10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
	¹¹ MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
	¹² TYPE(MPI_File), INTENT(IN) :: fh
	¹³ TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf
	¹⁴ INTEGER, INTENT(IN) :: count
	¹⁵ TYPE(MPI_Datatype), INTENT(IN) :: datatype
	¹⁶ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	17
	MPI_File_read_ordered_end(fh, buf, status, ierror) BIND(C)
	19 TYPE(MPI_File), INTENT(IN) :: fh
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf
	TYPE(MPI_Status) :: status
+:-l+ 040T	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	²³ MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) BIND(C)
	TYPE(MPI_File), INTENT(IN) :: fh
	TYPE(*), DIMENSION() :: buf
	INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype
	TYPE(MPI_Status) :: status
	²⁹ INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	INTEGER, OFITONAL, INTENT(OOT) TETTOT
	MPI_File_read_shared(fh, buf, count, datatype, status, ierror) BIND(C)
	TYPE(MPI_File), INTENT(IN) :: fh
	TYPE(*), DIMENSION() :: buf
	INTEGER, INTENT(IN) :: count
	<pre>35 TYPE(MPI_Datatype), INTENT(IN) :: datatype</pre>
	36 TYPE(MPI_Status) :: status
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	29
	MP1_File_seek(in, offset, whence, ierror) BIND(C)
	TYPE(MPI_File), INTENT(IN) :: In
	INIEGER(KIND=MPI_UFFSEI_KIND), INIENI(IN) :: OIISET
	INTEGER, INTENT(IN) :: whence
ticket-248T.	INTEGER, UPTIUNAL, INTENT(UUT) :: lerror
	MPI_File_seek_shared(fh, offset, whence, ierror) BIND(C)
	TYPE(MPI_File), INTENT(IN) :: fh
	46 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
	INTEGER, INTENT(IN) :: whence
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror

	1
MPI_File_set_atomicity(fh, flag, ierror) BIND(C)	$_2$ ticket-248T.
TYPE(MPI_File), INTENT(IN) :: fh	3
LOGICAL, INTENT(IN) :: flag	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$_{6}^{5}$ ticket-248T.
MPI_File_set_info(fh, info, ierror) BIND(C)	0
TYPE(MPI_File), INTENT(IN) :: fh	7
TYPE(MPI_Info), INTENT(IN) :: info	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9 10 1 0 40T
MPI_File_set_size(fh, size, ierror) BIND(C)	10 ticket-248T.
TYPE(MPI_File), INTENT(IN) :: fh	12
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
	$_{15}$ ticket-248T.
MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)	16
TYPE(MPI_File), INTENT(IN) :: fh	17
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype	18
CHARACTER(LEN=*), INTENT(IN) :: datarep	19
TYPE(MPI_Info), INTENT(IN) :: info	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
	22 ticket-248T.
MPI_File_sync(fh, ierror) BIND(C)	23
TYPE(MPI_File), INTENT(IN) :: fh	24 25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$_{26}^{25}$ ticket-248T.
<pre>MPI_File_write_all_begin(fh, buf, count, datatype, ierror) BIND(C)</pre>	27
TYPE(MPI_File), INTENT(IN) :: fh	28
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	29
INTEGER, INTENT(IN) :: count	30
TYPE(MPI_Datatype), INTENT(IN) :: datatype	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 ticket-248T.
MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)	33
TYPE(MPI_File), INTENT(IN) :: fh	34
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	35
TYPE(MPI_Status) :: status	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	³⁷ ₃₈ ticket-248T.
MPI_File_write_all(fh, buf, count, datatype, status, ierror) BIND(C)	39
TYPE(MPI_File), INTENT(IN) :: fh	40
TYPE(*), DIMENSION(), INTENT(IN) :: buf	41
INTEGER, INTENT(IN) :: count	42
TYPE(MPI_Datatype), INTENT(IN) :: datatype	43
TYPE(MPI_Status) :: status	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45 ticket-248T.
MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)	46
BIND(C)	47
	48

	1	TYPE(MPI_File), INTENT(IN) :: fh
	2	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
	3	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	4	INTEGER, INTENT(IN) :: count
	5	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- $248T$.	•	
	8	<pre>MPI_File_write_at_all_end(fh, buf, status, ierror) BIND(C)</pre>
	9	TYPE(MPI_File), INTENT(IN) :: fh
	10	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf</pre>
	11	TYPE(MPI_Status) :: status
ticket-248T.	12	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
		MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
	14	BIND(C)
	15	TYPE(MPI_File), INTENT(IN) :: fh
	16	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
	17	TYPE(*), DIMENSION(), INTENT(IN) :: buf
	18	INTEGER, INTENT(IN) :: count
	19	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	20	TYPE(MPI_Status) :: status
	21	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.		NDT File mite at (fb affact buf sound betating at the inner) DIND(0)
	23	MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
	24	TYPE(MPI_File), INTENT(IN) :: fh
	25	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
	26	TYPE(*), DIMENSION(), INTENT(IN) :: buf
	27	INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype
	28	TYPE(MPI_Datatype), INTENT(IN) datatype TYPE(MPI_Status) :: status
	29	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket- 2481 .	30	INTEGER, OFITOWAL, INTENT(001) TEITOT
		<pre>MPI_File_write(fh, buf, count, datatype, status, ierror) BIND(C)</pre>
	32	TYPE(MPI_File), INTENT(IN) :: fh
	33	<pre>TYPE(*), DIMENSION(), INTENT(IN) :: buf</pre>
	34	INTEGER, INTENT(IN) :: count
	35	TYPE(MPI_Datatype), INTENT(IN) :: datatype
	36	TYPE(MPI_Status) :: status
ticket-248T.	37	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
		MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
	39 40	TYPE(MPI_File), INTENT(IN) :: fh
	40	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf
	41	INTEGER, INTENT(IN) :: count
	42	TYPE(MPI_Datatype), INTENT(IN) :: datatype
		INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T.	44 45	
	45 46	MPI_File_write_ordered_end(fh, buf, status, ierror) BIND(C)
	40	TYPE(MPI_File), INTENT(IN) :: fh
	48	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf</pre>

```
1
    TYPE(MPI_Status) :: status
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   _{3} ticket-248T.
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                   4
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   5
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   6
    INTEGER, INTENT(IN) :: count
                                                                                   7
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   8
    TYPE(MPI_Status) :: status
                                                                                   9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  ^{10} ticket-248T.
                                                                                  11
MPI_File_write_shared(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                  12
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  13
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                  14
    INTEGER, INTENT(IN) :: count
                                                                                   15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   16
    TYPE(MPI_Status) :: status
                                                                                   17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18 ticket-248T.
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
                                                                                  19
              dtype_file_extent_fn, extra_state, ierror) BIND(C)
                                                                                  20
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                  21
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
                                                                                  22
    PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
                                                                                  23
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
                                                                                  24
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   26
                                                                                  27
                                                                                  28
A.3.12 Language Bindings Fortran 2008 Bindings
                                                                                  ^{29} ticket-248T.
MPI_F_sync_reg(buf) BIND(C)
                                                                                  30
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
                                                                                  _{32} ticket-248T.
MPI_Sizeof(x, size, ierror) BIND(C)
                                                                                  33
    TYPE(*), DIMENSION(..) :: x
                                                                                  34
    INTEGER, INTENT(OUT) :: size
                                                                                  35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  ^{36} ticket-248T.
MPI_Status_f082f(f08_status, f_status, ierror) BIND(C)
                                                                                  37
    TYPE(MPI_Status), INTENT(IN) :: f08_status
                                                                                  38
    INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   40
                                                                                  _{41} ticket-248T.
MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C)
                                                                                  42
    INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
                                                                                  43
    TYPE(MPI_Status), INTENT(OUT) :: f08_status
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  ^{45} ticket-248T.
MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
                                                                                  46
    INTEGER, INTENT(IN) :: p, r
                                                                                   47
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                   48
```

ticket-248T. $_{2}^{1}$	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
4	INTEGER, INTENT(IN) :: r
5	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
ticket-248T. 6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-2401. 7	<pre>MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C)</pre>
8	INTEGER, INTENT(IN) :: p, r
9	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
10	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 11	MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C)
12	INTEGER, INTENT(IN) :: typeclass, size
13 14	TYPE(MPI_Datatype), INTENT(OUT) :: datatype
14	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16	
17	
18	A.3.13 Profiling Interface Fortran 2008 Bindings
19	A.3.14 Deprecated Fortran 2008 Bindings
20	
21	
22	
23	
24 25	
25	
20	
28	
29	
30	
31	
32	
33	
34	
35 36	
30 37	
38	
39	
40	
41	
42	
43	
44	
45	
46 47	
47	
UF.	

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	805
A.4 Fortran Bindings with mpif.h or the mpi Module	1
A 4.1 Drint to Drint Communication Fouture Dindings	2
A.4.1 Point-to-Point Communication Fortran Bindings	3
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	4
<type> BUF(*)</type>	6
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	7
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	8
<type> BUF(*)</type>	9
INTEGER [REQUEST,]COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	10 ticket250-V.
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)	11
<pre><type> BUFFER(*)</type></pre>	12
INTEGER SIZE, IERROR	13
	14 15
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR) <type> BUFFER_ADDR(*)</type>	15
INTEGER SIZE, IERROR	17
	18
MPI_CANCEL(REQUEST, IERROR)	19
INTEGER REQUEST, IERROR	20
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)	21
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	22
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	23
<pre><type> BUF(*)</type></pre>	24
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	25 26
	20
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS(MPI_STATUS_SIZE),	28
INTEGER SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	29
	30
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)	31
<pre><type> BUF(*) INTEGED COUNT DATATABLE MEGGAGE DECUEGT IEDDOD</type></pre>	32
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR	33
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)	34
LOGICAL FLAG	35
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	36
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	37 38
<pre><type> BUF(*)</type></pre>	39
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	40
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	41
<pre><type> BUF(*)</type></pre>	42
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	43
	44
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	45
<pre><type> BUF(*) INTEGED COUNT DATATYDE DEST TAC COMM DECUEST LEDDOD</type></pre>	46
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	47
	48

```
1
               MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
           \mathbf{2}
                    <type> BUF(*)
           3
                    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
           4
               MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
           5
                    INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
           6
           7
               MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
           8
                    <type> BUF(*)
           9
                    INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
           10
               MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
          11
                    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
          12
          13
               MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
          14
                    <type> BUF(*)
          15
                    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
          16
                    IERROR
          17
               MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
           18
                    <type> BUF(*)
          19
                    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
          20
          21
               MPI_REQUEST_FREE(REQUEST, IERROR)
          22
                    INTEGER REQUEST, IERROR
          23
               MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
          24
                    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
          25
                    LOGICAL FLAG
          26
          27
               MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
          28
                    <type> BUF(*)
          29
                    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
          30
               MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
          ^{31}
                    <type> BUF(*)
          32
                    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
          33
          34
               MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
          35
                    <type> BUF(*)
          36
                    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
          37
               MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
          38
                    <type> BUF(*)
          39
ticket250-V. 40
                    INTEGER [REQUEST, ] COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
          41
               MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
          42
                             COMM, STATUS, IERROR)
          43
                    <type> BUF(*)
          44
                    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
          45
                    STATUS(MPI_STATUS_SIZE), IERROR
          46
          47
               MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
                             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
          48
```

<type> SENDBUF(*), RECVBUF(*) 1 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, $\mathbf{2}$ SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 5<type> BUF(*) 6 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 7 8 MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 9 <type> BUF(*) 10 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 11 MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR) 12INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR 13 14MPI_START(REQUEST, IERROR) 15INTEGER REQUEST, IERROR 16MPI TESTALL(COUNT, ARRAY OF REQUESTS, FLAG, ARRAY OF STATUSES, IERROR) 17LOGICAL FLAG 18 INTEGER COUNT, ARRAY_OF_REQUESTS(*), 19 ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 2021MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR) 22 LOGICAL FLAG 23INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), 24IERROR 25MPI_TEST_CANCELLED(STATUS, FLAG, IERROR) 26LOGICAL FLAG 27INTEGER STATUS(MPI_STATUS_SIZE), IERROR 28 29MPI_TEST(REQUEST, FLAG, STATUS, IERROR) 30 LOGICAL FLAG 31INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR 32 MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 33 ARRAY_OF_STATUSES, IERROR) 34 INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 35 ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 36 37 MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) 38 INTEGER COUNT, ARRAY_OF_REQUESTS(*) 39 INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 40MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR) 41 INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), 42IERROR 43 44MPI_WAIT(REQUEST, STATUS, IERROR) 45INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR 46MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 47ARRAY_OF_STATUSES, IERROR) 48

```
1
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
\mathbf{2}
         ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
3
4
     A.4.2 Datatypes Fortran Bindings
5
6
    MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
7
         <type> LOCATION(*)
8
         INTEGER IERROR
9
         INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
10
     MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
11
         INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
12
13
     MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
14
         INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
15
         INTEGER (KIND=MPI_COUNT_KIND) COUNT
16
     MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
17
                   POSITION, IERROR)
18
         INTEGER INCOUNT, DATATYPE, IERROR
19
         INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
20
         CHARACTER*(*) DATAREP
21
         <type> INBUF(*), OUTBUF(*)
22
23
     MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
24
         INTEGER INCOUNT, DATATYPE, IERROR
25
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
26
         CHARACTER*(*) DATAREP
27
     MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
28
         <type> INBUF(*), OUTBUF(*)
29
         INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
30
31
    MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
32
         INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
33
34
    MPI_TYPE_COMMIT(DATATYPE, IERROR)
35
         INTEGER DATATYPE, IERROR
36
     MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
37
         INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
38
39
     MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
40
                   ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
41
                   OLDTYPE, NEWTYPE, IERROR)
42
         INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
43
         ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
44
     MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
45
                   OLDTYPE, NEWTYPE, IERROR)
46
         INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
47
         INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
48
```

MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,	1
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	2
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR	3
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	4 5
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,	6
IERROR)	7
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	8
INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE	9
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	10
OLDTYPE, NEWTYPE, IERROR)	11
INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,	12
NEWTYPE, IERROR	13
	14
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)	15
INTEGER OLDTYPE, NEWTYPE, IERROR	16
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	17
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,	18
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)	19
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,	20
IERROR	21
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	22
	23
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,	24
ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),	25
ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR	26
ARRAI_OF_SIARIS(*), ORDER, OLDIIFE, NEWIIFE, IERROR	27
MPI_TYPE_DUP(<mark>OLD</mark> TYPE, NEWTYPE, IERROR)	28 ticket252-W.
INTEGER OLDTYPE, NEWTYPE, IERROR	29 ticket252-W.
MPI_TYPE_FREE(DATATYPE, IERROR)	30 31
INTEGER DATATYPE, IERROR	32
	33
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	34
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	35
IERROR)	36
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	37
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	38
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)	39
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,	40
COMBINER, IERROR)	41
INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,	42
IERROR	43
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)	44
INTEGER DATATYPE, IERROR	45
INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT	46
	47
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)	48

1 INTEGER DATATYPE, IERROR $\mathbf{2}$ INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT 3 MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, 4 OLDTYPE, NEWTYPE, IERROR) 5INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), 6 OLDTYPE, NEWTYPE, IERROR 7 8 MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) 9 INTEGER DATATYPE, SIZE, IERROR 10 MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) 11 INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR 1213MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, 14DATATYPE, IERROR) 15INTEGER OUTCOUNT, DATATYPE, IERROR 16INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION 17CHARACTER*(*) DATAREP 18 <type> INBUF(*), OUTBUF(*) 19 MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, 20IERROR) 21<type> INBUF(*), OUTBUF(*) 22 INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR 232425A.4.3 Collective Communication Fortran Bindings 26MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 27COMM, IERROR) 28<type> SENDBUF(*), RECVBUF(*) 29 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 30 31MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 32 RECVTYPE, COMM, IERROR) 33 <type> SENDBUF(*), RECVBUF(*) 34 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 35 IERROR 36 37 MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) 38INTEGER COUNT, DATATYPE, OP, COMM, IERROR 39 40MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 41 COMM, IERROR) 42<type> SENDBUF(*), RECVBUF(*) 43 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 44 45MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 46RDISPLS, RECVTYPE, COMM, IERROR) 47<type> SENDBUF(*), RECVBUF(*) 48

INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 1 2 RECVTYPE, COMM, IERROR MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR) 5 <type> SENDBUF(*), RECVBUF(*) 6 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 7 RDISPLS(*), RECVTYPES(*), COMM, IERROR 9 MPI_BARRIER(COMM, IERROR) 10 INTEGER COMM, IERROR 11 MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) 12<type> BUFFER(*) 13 INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR 1415MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 16 <type> SENDBUF(*), RECVBUF(*) 17INTEGER COUNT, DATATYPE, OP, COMM, IERROR 18 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 19 ROOT, COMM, IERROR) 20<type> SENDBUF(*), RECVBUF(*) 21INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 22 23 MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 24RECVTYPE, ROOT, COMM, IERROR) 25<type> SENDBUF(*), RECVBUF(*) 26INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 27COMM, IERROR 28 MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 29 COMM, REQUEST, IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 32 33 MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 34 RECVTYPE, COMM, REQUEST, IERROR) 35 <type> SENDBUF(*), RECVBUF(*) 36 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 37 REQUEST, IERROR 38 MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 39 IERROR) 40<type> SENDBUF(*), RECVBUF(*) 41 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 4243 MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 44COMM, REQUEST, IERROR) 45<type> SENDBUF(*), RECVBUF(*) 46INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 47 MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 48

1RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) $\mathbf{2}$ <type> SENDBUF(*), RECVBUF(*) 3 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 4 RECVTYPE, COMM, REQUEST, IERROR 5MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 6 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 7 <type> SENDBUF(*), RECVBUF(*) 8 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 9 RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR 10 11 MPI_IBARRIER(COMM, REQUEST, IERROR) 12INTEGER COMM, REQUEST, IERROR 13 MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR) 14 <type> BUFFER(*) 15INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR 1617MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 18 <type> SENDBUF(*), RECVBUF(*) 19 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 20MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 21ROOT, COMM, REQUEST, IERROR) 22 <type> SENDBUF(*), RECVBUF(*) 23INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 24IERROR 2526MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 27RECVTYPE, ROOT, COMM, REQUEST, IERROR) 28 <type> SENDBUF(*), RECVBUF(*) 29INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 30 COMM, REQUEST, IERROR 31MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 32 REQUEST, IERROR) 33 <type> SENDBUF(*), RECVBUF(*) 34 INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR 35 36 MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 37 REQUEST, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR 40MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, 41 IERROR) 42<type> SENDBUF(*), RECVBUF(*) 43 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR 44 45MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 46<type> SENDBUF(*), RECVBUF(*) 47INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 48

MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	1
ROOT, COMM, REQUEST, IERROR)	2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	4
IERROR	5
MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	6
RECVTYPE, ROOT, COMM, REQUEST, IERROR)	7 8
<type> SENDBUF(*), RECVBUF(*)</type>	9
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	9 10
COMM, REQUEST, IERROR	11
	12
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)	13
LOGICAL COMMUTE	14
INTEGER OP, IERROR	15
MPI_OP_CREATE([FUNCTION] USER_FN, COMMUTE, OP, IERROR)	$_{16}^{16}$ ticket252-W.
EXTERNAL [FUNCTION]USER_FN	$_{17}$ ticket252-W.
LOGICAL COMMUTE	18
INTEGER OP, IERROR	19
MPI_OP_FREE(OP, IERROR)	20
INTEGER OP, IERROR	21
INTEGER OF, IERROR	22
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)	$_{23}$ ticket 250-V.
<type> INBUF(*), INOUTBUF(*)</type>	24
INTEGER COUNT, DATATYPE, OP, IERROR	25
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,	26
IERROR)	27
<type> SENDBUF(*), RECVBUF(*)</type>	28
INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR	29
	30
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,	31
IERROR)	32
<type> SENDBUF(*), RECVBUF(*)</type>	33
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR	34
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)	35
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	36
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR	37
	38
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	39
<type> SENDBUF(*), RECVBUF(*)</type>	40
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	41
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	42
ROOT, COMM, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR	45
	46
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)	47 48
	40

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
3
         COMM, IERROR
4
5
     A.4.4 Groups, Contexts, Communicators, and Caching Fortran Bindings
6
\overline{7}
     MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
8
         INTEGER COMM1, COMM2, RESULT, IERROR
9
     MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
10
         INTEGER COMM, GROUP, NEWCOMM, IERROR
11
12
     MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
13
                   EXTRA_STATE, IERROR)
14
         EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
15
         INTEGER COMM_KEYVAL, IERROR
16
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
17
     MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
18
         INTEGER COMM, COMM_KEYVAL, IERROR
19
20
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
21
         INTEGER COMM, NEWCOMM, IERROR
22
     MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
23
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
24
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
25
26
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT
27
         LOGICAL FLAG
28
29
     MPI_COMM_FREE(COMM, IERROR)
30
         INTEGER COMM, IERROR
^{31}
     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
32
33
         INTEGER COMM_KEYVAL, IERROR
34
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
35
         INTEGER COMM, COMM_KEYVAL, IERROR
36
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
37
         LOGICAL FLAG
38
39
     MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
40
         INTEGER COMM, RESULTLEN, IERROR
41
         CHARACTER*(*) COMM_NAME
42
     MPI_COMM_GROUP(COMM, GROUP, IERROR)
43
         INTEGER COMM, GROUP, IERROR
44
45
     MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
46
47
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
48
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
```

ATTRIBUTE_VAL_OUT LOGICAL FLAG	$\frac{1}{2}$
MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,	3
IERROR)	4
INTEGER COMM, COMM_KEYVAL, IERROR	5 6
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	7
MPI_COMM_RANK(COMM, RANK, IERROR)	8
INTEGER COMM, RANK, IERROR	9
	10
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR	11
INTEGER COMM, GROOF, IERROR	12
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)	13 14
INTEGER COMM, SIZE, IERROR	14
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)	16
INTEGER COMM, COMM_KEYVAL, IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	18
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)	19
INTEGER COMM, IERROR	20
CHARACTER*(*) COMM_NAME	21
MPI_COMM_SIZE(COMM, SIZE, IERROR)	22 23
INTEGER COMM, SIZE, IERROR	23 24
	25
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)	26
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	27
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)	28
INTEGER COMM, IERROR	29
LOGICAL FLAG	30
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)	31
INTEGER GROUP1, GROUP2, RESULT, IERROR	32 33
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)	34
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	35
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)	36
INTEGER GROUP, N, RANKS, NEWGROUP, IERROR	37
	38
MPI_GROUP_FREE(GROUP, IERROR)	39
INTEGER GROUP, IERROR	40 41
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)	41 42
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	43
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)	44
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	45
	46
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR	47
	48

1 2	MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
3 4 5	MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR
6 7 8	MPI_GROUP_SIZE(GROUP, SIZE, IERROR) INTEGER GROUP, SIZE, IERROR
9 10	MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
11 12 13	MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
13 14 15 16 17	MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR) INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR
ticket250-V. ¹⁸ ticket250-V. ¹⁹ 20 21	MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH
22 23 24 25 26	<pre>MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL, EXTRA_STATE, IERROR) EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN INTEGER TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</pre>
ticket252-W. $^{27}_{28}$ ticket252-W. $^{28}_{29}$	MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR
29 30 31 32 33 34 35	<pre>MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>
36 37 38	MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR) INTEGER TYPE_KEYVAL, IERROR
ticket252-W. 39 ticket252-W. 40 41 42	MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL LOGICAL FLAG
ticket252-W. $^{43}_{44}$ ticket252-W. $^{43}_{45}_{45}$	MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR CHARACTER*(*) TYPE_NAME
47 48	MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)

INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	1
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	2
ATTRIBUTE_VAL_OUT	3
LOGICAL FLAG	4 5
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)	$_{6}^{\circ}$ ticket252-W.
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	$\frac{7}{8}$ ticket252-W.
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	9
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)	10 ticket252-W.
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	11 ticket252-W.
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	12
	13
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)	$_{14}$ ticket252-W.
INTEGER DATATYPE, IERROR	$_{15}$ ticket252-W.
CHARACTER*(*) TYPE_NAME	16
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	17
EXTRA_STATE, IERROR)	18
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	19
INTEGER WIN_KEYVAL, IERROR	20 21
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	21 22
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	23
INTEGER WIN, WIN_KEYVAL, IERROR	24
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	25
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	26
INTEGER OLDWIN, WIN_KEYVAL, IERROR	27
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	28
ATTRIBUTE_VAL_OUT	29
LOGICAL FLAG	30
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)	31
INTEGER WIN_KEYVAL, IERROR	32
	33 34
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	35
INTEGER WIN, WIN_KEYVAL, IERROR	36
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	37
LOGICAL FLAG	38
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)	39
INTEGER WIN, RESULTLEN, IERROR	40
CHARACTER*(*) WIN_NAME	41
MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	42
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	43
INTEGER OLDWIN, WIN_KEYVAL, IERROR	44
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	45 46
ATTRIBUTE_VAL_OUT	46
LOGICAL FLAG	48

```
1
    MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
\mathbf{2}
         INTEGER WIN, WIN_KEYVAL, IERROR
3
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
4
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
5
         INTEGER WIN, WIN_KEYVAL, IERROR
6
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
7
8
     MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
9
         INTEGER WIN, IERROR
10
         CHARACTER*(*) WIN_NAME
11
12
     A.4.5 Process Topologies Fortran Bindings
13
14
    MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
15
         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
16
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
17
         INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
18
         LOGICAL PERIODS(*), REORDER
19
20
     MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
21
         INTEGER COMM, NDIMS, IERROR
22
    MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
23
         INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
^{24}
         LOGICAL PERIODS(*)
25
26
     MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
27
         INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
28
         LOGICAL PERIODS(*)
29
30
    MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
         INTEGER COMM, COORDS(*), RANK, IERROR
31
32
     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
33
         INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
34
35
     MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
36
         INTEGER COMM, NEWCOMM, IERROR
37
         LOGICAL REMAIN_DIMS(*)
38
    MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
39
         INTEGER NNODES, NDIMS, DIMS(*), IERROR
40
^{41}
     MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
42
                   OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
43
                   COMM_DIST_GRAPH, IERROR)
44
         INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
45
             DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
46
         LOGICAL REORDER
47
48
```

MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	1					
INFO, REORDER, COMM_DIST_GRAPH, IERROR)	2					
<pre>INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),</pre>	3					
WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR	4					
LOGICAL REORDER	5					
	6					
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,	7					
MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)	8					
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,						
DESTINATIONS(*), DESTWEIGHTS(*), IERROR	10					
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)	11					
INTEGER COMM, INDEGREE, OUTDEGREE, IERROR	12					
LOGICAL WEIGHTED	13					
	14					
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,	15					
IERROR)	16					
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR	17					
LOGICAL REORDER	18					
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)	19					
INTEGER COMM, NNODES, NEDGES, IERROR	20					
INIEGEN COMM, MNODED, MEDGED, IEMION	21					
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)	22					
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	23					
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)						
INTEGER COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR	25					
INTEGEN COMM, NNODED, INDER(*), EDGED(*), NEWNANA, IEMMON	26					
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)	27					
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR						
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)	29					
	30					
INTEGER COMM, RANK, NNEIGHBORS, IERROR						
MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	32					
RECVTYPE, COMM, REQUEST, IERROR)	33					
<type> SENDBUF(*), RECVBUF(*)</type>	34					
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	35					
	36					
MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	37					
DISPLS, RECVTYPE, COMM, REQUEST, IERROR)	38					
<pre><type> SENDBUF(*), RECVBUF(*) INTEGED GENDGOUNT GENDTYDE DEGUGOUNTG(*) DIGDLG(*) DEGUTYDE GOMM</type></pre>	39					
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	40					
REQUEST, IERROR	41					
MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	42					
RECVTYPE, COMM, REQUEST, IERROR)	43					
<type> SENDBUF(*), RECVBUF(*)</type>	44					
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	45					
	46					
MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	47					
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)	48					

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
3
         RECVTYPE, COMM, REQUEST, IERROR
4
     MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
5
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
6
         <type> SENDBUF(*), RECVBUF(*)
7
         INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
8
         INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
9
         REQUEST, IERROR
10
11
     MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
12
                  RECVTYPE, COMM, IERROR)
13
         <type> SENDBUF(*), RECVBUF(*)
14
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
15
     MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
16
                   DISPLS, RECVTYPE, COMM, IERROR)
17
         <type> SENDBUF(*), RECVBUF(*)
18
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
19
         IERROR
20
21
     MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
22
                   RECVTYPE, COMM, IERROR)
23
         <type> SENDBUF(*), RECVBUF(*)
^{24}
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
25
     MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
26
                   RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
27
         <type> SENDBUF(*), RECVBUF(*)
28
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
29
         RECVTYPE, COMM, IERROR
30
31
     MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
32
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
33
         <type> SENDBUF(*), RECVBUF(*)
34
         INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
35
         INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
36
         IERROR
37
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
38
         INTEGER COMM, STATUS, IERROR
39
40
41
     A.4.6 MPI Environmental Management Fortran Bindings
42
    DOUBLE PRECISION MPI_WTICK()
43
44
    DOUBLE PRECISION MPI_WTIME()
45
46
    MPI_ABORT(COMM, ERRORCODE, IERROR)
47
         INTEGER COMM, ERRORCODE, IERROR
48
```

MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR	1 2
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)	3 4
INTEGER ERRORCLASS, ERRORCODE, IERROR	5
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)	6
INTEGER ERRORCODE, IERROR	7
CHARACTER*(*) STRING	8
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)	9
INTEGER INFO, IERROR	10
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	11
	12 13
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)	13
INTEGER COMM, ERRORCODE, IERROR	15
MPI_COMM_CREATE_ERRHANDLER([FUNCTION]COMM_ERRHANDLER_FN, ERRHANDLER,	$_{16}$ ticket252-W.
IERROR)	17
EXTERNAL [FUNCTION] COMM_ERRHANDLER_FN	$_{18}$ ticket252-W.
INTEGER ERRHANDLER, IERROR	19
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)	20
INTEGER COMM, ERRHANDLER, IERROR	21
	22
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)	23
INTEGER COMM, ERRHANDLER, IERROR	24
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)	25
INTEGER ERRHANDLER, IERROR	26
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)	27
INTEGER ERRORCODE, ERRORCLASS, IERROR	28 29
	30
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)	31
INTEGER ERRORCODE, RESULTLEN, IERROR	32
CHARACTER*(*) STRING	33
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)	34
INTEGER FH, ERRORCODE, IERROR	35
MPI_FILE_CREATE_ERRHANDLER([FUNCTION]FILE_ERRHANDLER_FN, ERRHANDLER,	36 ticket252-W.
IERROR)	37
EXTERNAL [FUNCTION] FILE_ERRHANDLER_FN	$^{38}_{39}$ ticket252-W.
INTEGER ERRHANDLER, IERROR	39 40
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	40
INTEGER FILE, ERRHANDLER, IERROR	41 42
	43
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	44
INTEGER FILE, ERRHANDLER, IERROR	45
MPI_FINALIZED(FLAG, IERROR)	46
LOGICAL FLAG	47
INTEGER IERROR	48

	2	MPI_FINALIZE(IERROR) INTEGER IERROR
	3 4 5 6	MPI_FREE_MEM(BASE, IERROR) <type> BASE(*) INTEGER IERROR</type>
	8 9	MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR) CHARACTER*(*) VERSION INTEGER RESULTLEN,IERROR
	10 11 12 13	MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR) CHARACTER*(*) NAME INTEGER RESULTLEN,IERROR
	14 15 16	MPI_GET_VERSION(VERSION, SUBVERSION, IERROR) INTEGER VERSION, SUBVERSION, IERROR
	17 18 19	MPI_INITIALIZED(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR
	20 21 22	MPI_INIT(IERROR) INTEGER IERROR
	23 24	MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR) INTEGER WIN, ERRORCODE, IERROR
ticket252-W. ticket252-W.	25 26 27 28	MPI_WIN_CREATE_ERRHANDLER([FUNCTION]WIN_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL [FUNCTION]WIN_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR
		MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR
	31 32 33 34	MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR
	35	A.4.7 The Info Object Fortran Bindings
	36 37 38	MPI_INFO_CREATE(INFO, IERROR) INTEGER INFO, IERROR
	39 40 41	MPI_INFO_DELETE(INFO, KEY, IERROR) INTEGER INFO, IERROR CHARACTER*(*) KEY
	42 43 44	MPI_INFO_DUP(INFO, NEWINFO, IERROR) INTEGER INFO, NEWINFO, IERROR
	45 46 47	MPI_INFO_FREE(INFO, IERROR) INTEGER INFO, IERROR
	48	MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)

```
1
    INTEGER INFO, VALUELEN, IERROR
                                                                                    2
    CHARACTER*(*) KEY, VALUE
                                                                                    3
    LOGICAL FLAG
                                                                                    4
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
                                                                                    5
    INTEGER INFO, NKEYS, IERROR
                                                                                    6
                                                                                    7
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
                                                                                    8
    INTEGER INFO, N, IERROR
                                                                                    9
    CHARACTER*(*) KEY
                                                                                    10
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
                                                                                    11
    INTEGER INFO, VALUELEN, IERROR
                                                                                   12
    LOGICAL FLAG
                                                                                    13
    CHARACTER*(*) KEY
                                                                                   14
                                                                                    15
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
                                                                                    16
    INTEGER INFO, IERROR
                                                                                    17
    CHARACTER*(*) KEY, VALUE
                                                                                    18
                                                                                   19
A.4.8 Process Creation and Management Fortran Bindings
                                                                                   20
                                                                                   21
MPI_CLOSE_PORT(PORT_NAME, IERROR)
                                                                                   22
    CHARACTER*(*) PORT_NAME
                                                                                   23
    INTEGER IERROR
                                                                                   24
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
                                                                                   25
    CHARACTER*(*) PORT_NAME
                                                                                   26
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
                                                                                   27
                                                                                   28
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
                                                                                   29
    CHARACTER*(*) PORT_NAME
                                                                                   30
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
                                                                                    31
MPI_COMM_DISCONNECT(COMM, IERROR)
                                                                                    32
    INTEGER COMM, IERROR
                                                                                   33
                                                                                   34
MPI_COMM_GET_PARENT(PARENT, IERROR)
                                                                                   35
    INTEGER PARENT, IERROR
                                                                                   36
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
                                                                                   37
    INTEGER FD, INTERCOMM, IERROR
                                                                                   38
                                                                                   39
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
                                                                                    40
              ARRAY_OF_ERRCODES, IERROR)
                                                                                   41
    CHARACTER*(*) COMMAND, ARGV(*)
                                                                                   42
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
                                                                                   43
    IERROR
                                                                                   44
                                                                                   45
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
                                                                                   46
              ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
                                                                                   47
              ARRAY_OF_ERRCODES, IERROR)
                                                                                    48
```

```
1
         INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
\mathbf{2}
         INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
3
         CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
4
     MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
5
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
6
         INTEGER INFO. IERROR
7
8
     MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
9
         CHARACTER*(*) PORT_NAME
10
         INTEGER INFO, IERROR
11
     MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
12
         INTEGER INFO, IERROR
13
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
14
15
     MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
16
         INTEGER INFO, IERROR
17
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
18
19
     A.4.9 One-Sided Communications Fortran Bindings
20
21
     MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
22
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
23
         <type> ORIGIN_ADDR(*)
24
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
25
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
26
         TARGET_DATATYPE, OP, WIN, IERROR
27
     MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
28
                   TARGET_RANK, TARGET_DISP, WIN, IERROR)
29
         <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
30
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
31
         INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
32
33
     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
34
                   TARGET_DISP, OP, WIN, IERROR)
35
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
36
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
37
         INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
38
     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
39
40
                   RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
41
                   TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
42
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
43
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
44
45
         TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
46
     MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
47
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
48
```

1 <type> ORIGIN_ADDR(*) $\mathbf{2}$ INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 3 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR 4 5MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 6 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR) 7 <type> ORIGIN_ADDR(*) 8 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 9 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 10 TARGET_DATATYPE, WIN, IERROR 11 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 1213 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 14IERROR) 15<type> ORIGIN_ADDR(*) 16INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 17INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 18 TARGET_DATATYPE, OP, WIN, REQUEST, IERROR 19 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, 20RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, 21TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, 22 IERROR) 23<type> ORIGIN_ADDR(*), RESULT_ADDR(*) 24INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 25INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE, 26TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR 2728 MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 29TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, 30 IERROR) 31<type> ORIGIN_ADDR(*) 32 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 33 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 34 TARGET_DATATYPE, WIN, REQUEST, IERROR 35MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 36 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, 37 IERROR) 38 <type> ORIGIN_ADDR(*) 39 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 40INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 41 TARGET_DATATYPE, WIN, REQUEST, IERROR 4243 MPI_WIN_ALLOCATE_SHARED(SIZE, INFO, COMM, BASEPTR, WIN, IERROR) 44INTEGER INFO, COMM, WIN, IERROR 45INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 46MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 47INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 48

	1	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
	2 3	MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
	4	INTEGER WIN, IERROR
txx:12/9/11.	0	<type> [base]BASE(*) INTEGER (KIND=MPI_ADDRESS_[SIZE]KIND) [size]SIZE</type>
txx:8/16/11. txx:12/9/11.		
uxx.12/9/11.	8	MPI_WIN_COMPLETE(WIN, IERROR) INTEGER WIN, IERROR
	9 10	MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
	11	<type> BASE(*)</type>
	12	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
	13	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
	14 15	MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR) INTEGER INFO, COMM, WIN, IERROR
	16	
		MPI_WIN_DETACH(WIN, BASE, IERROR) INTEGER WIN, IERROR
txx:12/9/11.	18 19	<type> [base]BASE(*)</type>
	20	MPI_WIN_FENCE(ASSERT, WIN, IERROR)
	21	INTEGER ASSERT, WIN, IERROR
	22 23	MPI_WIN_FLUSH_ALL(WIN, IERROR)
	24	INTEGER WIN, IERROR
	25	MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
	26 27	INTEGER WIN, IERROR
	28	MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
	29	INTEGER RANK, WIN, IERROR
	30 31	MPI_WIN_FLUSH(RANK, WIN, IERROR)
	32	INTEGER RANK, WIN, IERROR
	33 34	MPI_WIN_FREE(WIN, IERROR) INTEGER WIN, IERROR
	35	
	36	MPI_WIN_GET_GROUP(WIN, GROUP, IERROR) INTEGER WIN, GROUP, IERROR
	37 38	MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
	39	INTEGER ASSERT, WIN, IERROR
	40	MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
	41 42	INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
	43	MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
	44	INTEGER GROUP, ASSERT, WIN, IERROR
	45 46	MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, BASEPTR, IERROR)
	47	INTEGER WIN, RANK, IERROR
	48	INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR	1 2		
MPI_WIN_SYNC(WIN, IERROR) INTEGER WIN, IERROR	3 4 5		
MPI_WIN_TEST(WIN, FLAG, IERROR) INTEGER WIN, IERROR LOGICAL FLAG	6 7 8		
MPI_WIN_UNLOCK_ALL(WIN, IERROR) INTEGER WIN, IERROR	9 10 11		
MPI_WIN_UNLOCK(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR	12 13 14		
MPI_WIN_WAIT(WIN, IERROR) INTEGER WIN, IERROR	14 15 16		
A.4.10 External Interfaces Fortran Bindings	17 18 19		
MPI_GREQUEST_COMPLETE(REQUEST, IERROR) INTEGER REQUEST, IERROR	20 21		
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR) INTEGER REQUEST, IERROR	22 23 24		
EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE	25 26 27		
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR) INTEGER REQUIRED, PROVIDED, IERROR	28 29		
MPI_IS_THREAD_MAIN(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR	30 31 32 33		
MPI_QUERY_THREAD(PROVIDED, IERROR) INTEGER PROVIDED, IERROR	34 35		
MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG	36 37 38 39		
MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	40 41 42		
MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER (KIND=MPI_COUNT_KIND) COUNT			
INIEGEN (NIND IN I_OUONI_NIND) OUONI	45 46 47		
	48		

```
1
     A.4.11 I/O Fortran Bindings
\mathbf{2}
     MPI_FILE_CLOSE(FH, IERROR)
3
         INTEGER FH, IERROR
4
\mathbf{5}
     MPI_FILE_DELETE(FILENAME, INFO, IERROR)
6
         CHARACTER*(*) FILENAME
7
         INTEGER INFO, IERROR
8
     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
9
         INTEGER FH, AMODE, IERROR
10
^{11}
     MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
12
         INTEGER FH, IERROR
13
         LOGICAL FLAG
14
     MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
15
         INTEGER FH, IERROR
16
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
17
18
     MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
19
         INTEGER FH, GROUP, IERROR
20
     MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
21
         INTEGER FH, INFO_USED, IERROR
22
23
     MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
^{24}
         INTEGER FH, IERROR
25
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
26
     MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
27
         INTEGER FH, IERROR
28
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
29
30
     MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
^{31}
         INTEGER FH, IERROR
32
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
33
34
     MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
         INTEGER FH, DATATYPE, IERROR
35
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
36
37
     MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
38
         INTEGER FH, ETYPE, FILETYPE, IERROR
39
         CHARACTER*(*) DATAREP
40
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
41
     MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
42
         <type> BUF(*)
43
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
44
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
46
     MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
47
         <type> BUF(*)
48
```

1 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 2 MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) <type> BUF(*) 4 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 56 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 7 <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 9 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 10MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 11 <type> BUF(*) 12INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 13 14MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 15<type> BUF(*) 16INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 17MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) 18 CHARACTER*(*) FILENAME 19 INTEGER COMM, AMODE, INFO, FH, IERROR 2021MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) 22 INTEGER FH, IERROR 23INTEGER(KIND=MPI_OFFSET_KIND) SIZE 24MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) 25<type> BUF(*) 26INTEGER FH, COUNT, DATATYPE, IERROR 2728 MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR) 29<type> BUF(*) 30 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 31MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 32 <type> BUF(*) 33 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 34 35 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 36 <type> BUF(*) 37 INTEGER FH, COUNT, DATATYPE, IERROR 38 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 39 MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR) 40<type> BUF(*) 41 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 4243 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 44<type> BUF(*) 45INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 46INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 47MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 48

```
1
         <type> BUF(*)
\mathbf{2}
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
3
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
4
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
5
         <type> BUF(*)
6
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
7
8
     MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
9
         <type> BUF(*)
10
         INTEGER FH, COUNT, DATATYPE, IERROR
11
    MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
12
         <type> BUF(*)
13
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
14
15
    MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
16
         <type> BUF(*)
17
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
18
    MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
19
         <type> BUF(*)
20
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
21
22
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
23
         INTEGER FH, WHENCE, IERROR
^{24}
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
25
     MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
26
         INTEGER FH, WHENCE, IERROR
27
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
28
29
     MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
30
         INTEGER FH, IERROR
^{31}
         LOGICAL FLAG
32
    MPI_FILE_SET_INFO(FH, INFO, IERROR)
33
         INTEGER FH, INFO, IERROR
34
35
    MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
36
         INTEGER FH, IERROR
37
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
38
39
     MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
         INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
40
         CHARACTER*(*) DATAREP
41
42
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
43
     MPI_FILE_SYNC(FH, IERROR)
44
         INTEGER FH, IERROR
45
46
    MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
47
         <type> BUF(*)
48
         INTEGER FH, COUNT, DATATYPE, IERROR
```

MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)	1 2
<type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type>	3
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	4 5
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>	6 7
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)	8
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type>	9 10
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	11 12
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)	13
<type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type>	14 15
MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	16
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>	17 18
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	19 20
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	20
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>	22 23
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	24
<pre>MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	25 26
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	27 28
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	29
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type>	30 31
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)	32
<type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type>	33 34
MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	35
<type> BUF(*)</type>	36 37
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	38 39
<pre>MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	40
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	41 42
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,	43
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR) CHARACTER*(*) DATAREP	44 45
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	46 47
INTEGER (KIND-MPI_ADDRESS_KIND) EXIRA_SIAIE INTEGER IERROR	47 48

```
1
                A.4.12 Language Bindings Fortran Bindings
           \mathbf{2}
                MPI_F_SYNC_REG(buf)
            3
                     <type> buf(*)
            4
           \mathbf{5}
                MPI_SIZEOF(X, SIZE, IERROR)
           6
                     <type> X
           7
                     INTEGER SIZE, IERROR
            8
                MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
           9
                     TYPE(MPI_Status) :: F08_STATUS
           10
                     INTEGER :: F_STATUS(MPI_STATUS_SIZE)
           11
                     INTEGER IERROR
           12
           13
                MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
           14
                     INTEGER :: F_STATUS(MPI_STATUS_SIZE)
           15
                     TYPE(MPI_Status) :: F08_STATUS
           16
                     INTEGER IERROR
           17
                MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
           18
                     INTEGER P, R, NEWTYPE, IERROR
           19
           20
                MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
           21
                     INTEGER R, NEWTYPE, IERROR
           22
                MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
           23
                     INTEGER P, R, NEWTYPE, IERROR
           ^{24}
           25
ticket252-W.
                MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
           26
ticket252-W.
                     INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
           27
           28
                A.4.13 Profiling Interface Fortran Bindings
           29
           30
                MPI_PCONTROL(LEVEL)
           ^{31}
                     INTEGER LEVEL
           32
           33
           34
                A.4.14 Deprecated Fortran Bindings
           35
                MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
           36
                     <type> LOCATION(*)
           37
                     INTEGER ADDRESS, IERROR
           38
           39
                MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
           40
                     INTEGER COMM, KEYVAL, IERROR
           41
                MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
           42
                     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
           43
                     LOGICAL FLAG
           44
           45
                MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
           46
                     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
           47
           48
```

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	833
MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG	1 2 3 4 5
MPI_ERRHANDLER_CREATE([FUNCTION]HANDLER_FN, ERRHANDLER, IERROR) EXTERNAL [FUNCTION]HANDLER_FN INTEGER ERRHANDLER, IERROR	$_7^6$ ticket252-W. $_8$ ticket252-W. $_9$
MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	10 11
MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	12 13 14
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR) EXTERNAL COPY_FN, DELETE_FN INTEGER KEYVAL, EXTRA_STATE, IERROR	15 16 17 18
MPI_KEYVAL_FREE(KEYVAL, IERROR) INTEGER KEYVAL, IERROR	19 20
MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG	21 22 23 24 25 26
MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR	27 28
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR) INTEGER DATATYPE, EXTENT, IERROR	29 30 31
<pre>MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,</pre>	32 33 34 35
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	36 37 38
MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERROR) INTEGER DATATYPE, DISPLACEMENT, IERROR	39 40
<pre>MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,</pre>	41 42 43 44
MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERROR) INTEGER DATATYPE, DISPLACEMENT, IERROR	45 46 47 48

1	SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
2	ATTRIBUTE_VAL_OUT, FLAG, IERR)
3	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4	ATTRIBUTE_VAL_OUT, IERR
5	LOGICAL FLAG
6	
7	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
8	INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
9	
10	
11	
12	
13	
14	
15	
16	
17 18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36 37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

A.5 C++ Bindings (deprecated)	1
A.5.1 Point-to-Point Communication C++ Bindings	2 3
namespace MPI {	4
<pre>{void Attach_buffer(void* buffer, int size)(binding deprecated, see Section 15.2) }</pre>	5 6 7
<pre>{void Comm::Bsend(const void* buf, int count, const Datatype& datatype,</pre>	8 9 10
<pre>{Prequest Comm::Bsend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>	11 12 13 14
<pre>{void Request::Cancel() const(binding deprecated, see Section 15.2) }</pre>	15
<pre>{int Detach_buffer(void*& buffer)(binding deprecated, see Section 15.2) }</pre>	16 17
<pre>{void Request::Free()(binding deprecated, see Section 15.2) }</pre>	18
<pre>{int Status::Get_count(const Datatype& datatype) const(binding deprecated,</pre>	19 20 21
<pre>{int Status::Get_error() const(binding deprecated, see Section 15.2) }</pre>	22
<pre>{int Status::Get_source() const(binding deprecated, see Section 15.2) }</pre>	23 24
<pre>{bool Request::Get_status() const(binding deprecated, see Section 15.2) }</pre>	25
<pre>{bool Request::Get_status(Status& status) const(binding deprecated, see Section 15.2) }</pre>	26 27 28
<pre>{int Status::Get_tag() const(binding deprecated, see Section 15.2) }</pre>	29
<pre>{Request Comm::Ibsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>	30 31 32 33
<pre>{bool Comm::Iprobe(int source, int tag) const(binding deprecated, see Section 15.2) }</pre>	34 35
<pre>{bool Comm::Iprobe(int source, int tag, Status& status) const(binding</pre>	36 37 38
<pre>{Request Comm::Irecv(void* buf, int count, const Datatype& datatype,</pre>	39 40 41
<pre>{Request Comm::Irsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>	41 42 43 44
<pre>{bool Status::Is_cancelled() const(binding deprecated, see Section 15.2) }</pre>	45 46
<pre>{Request Comm::Isend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated,</pre>	40 47 48

	see Section 15.2 }
{Request (<pre>Comm::Issend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>
$\{void Comr$	<pre>n::Probe(int source, int tag) const(binding deprecated, see Section 15.2) }</pre>
$\{void Comr$	<pre>n::Probe(int source, int tag, Status& status) const(binding deprecated, see Section 15.2) }</pre>
$\{Prequest$	<pre>Comm::Recv_init(void* buf, int count, const Datatype& datatype, int source, int tag) const(binding deprecated, see Section 15.2) }</pre>
$\{void Comr$	<pre>n::Recv(void* buf, int count, const Datatype& datatype, int source, int tag) const(binding deprecated, see Section 15.2) }</pre>
{void Comr	<pre>n::Recv(void* buf, int count, const Datatype& datatype, int source, int tag, Status& status) const(binding deprecated, see Section 15.2) }</pre>
{void Comr	<pre>n::Rsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>
$\{Prequest$	<pre>Comm::Rsend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>
$\{void Comr$	<pre>n::Send(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>
$\{Prequest$	<pre>Comm::Send_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const(binding deprecated, see Section 15.2) }</pre>
{void Comr	<pre>n::Sendrecv(const void *sendbuf, int sendcount, const Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount, const Datatype& recvtype, int source, int recvtag) const(binding deprecated, see Section 15.2) }</pre>
{void Comr	<pre>n::Sendrecv(const void *sendbuf, int sendcount, const Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount, const Datatype& recvtype, int source, int recvtag, Status& status) const(binding deprecated, see Section 15.2) }</pre>
{void Comr	<pre>n::Sendrecv_replace(void* buf, int count, const Datatype& datatype, int dest, int sendtag, int source, int recvtag) const(binding deprecated, see Section 15.2) }</pre>
{void Comr	<pre>n::Sendrecv_replace(void* buf, int count, const Datatype& datatype, int dest, int sendtag, int source, int recvtag, Status& status) const(binding deprecated, see</pre>

Section 15.2 }
<pre>{void Status::Set_error(int error)(binding deprecated, see Section 15.2) }</pre>
<pre>{void Status::Set_source(int source)(binding deprecated, see Section 15.2) }</pre>
<pre>{void Status::Set_tag(int tag)(binding deprecated, see Section 15.2) }</pre>
<pre>{void Comm::Ssend(const void* buf, int count, const Datatype& datatype,</pre>
<pre>{Prequest Comm::Ssend_init(const void* buf, int count, const</pre>
<pre>{static void Prequest::Startall(int count,</pre>
<pre>{void Prequest::Start()(binding deprecated, see Section 15.2) }</pre>
<pre>{static bool Request::Testall(int count, Request array_of_requests[],</pre>
<pre>{static bool Request::Testall(int count,</pre>
<pre>{static bool Request::Testany(int count, Request array_of_requests[],</pre>
<pre>{static bool Request::Testany(int count, Request array_of_requests[],</pre>
<pre>{bool Request::Test()(binding deprecated, see Section 15.2) }</pre>
{bool Request::Test(Status& status)(binding deprecated, see Section 15.2)}
<pre>{static int Request::Testsome(int incount, Request array_of_requests[],</pre>
<pre>{static int Request::Testsome(int incount, Request array_of_requests[],</pre>
<pre>{static void Request::Waitall(int count, Request array_of_requests[],</pre>
<pre>{static void Request::Waitall(int count,</pre>
<pre>{static int Request::Waitany(int count, Request array_of_requests[],</pre>
<pre>{static int Request::Waitany(int count,</pre>
<pre>{void Request::Wait(Status& status)(binding deprecated, see Section 15.2) } 44</pre>

```
1
       {static int Request::Waitsome(int incount, Request array_of_requests[],
2
                    int array_of_indices[], Status array_of_statuses[]) (binding
3
                    deprecated, see Section 15.2 }
4
       {static int Request::Waitsome(int incount, Request array_of_requests[],
5
                    int array_of_indices[]) (binding deprecated, see Section 15.2) }
6
7
       {void Request::Wait() (binding deprecated, see Section 15.2) }
8
9
     };
10
11
     A.5.2 Datatypes C++ Bindings
12
13
     namespace MPI {
14
15
       {void Datatype::Commit()(binding deprecated, see Section 15.2)}
16
       {Datatype Datatype::Create_contiguous(int count) const(binding deprecated,
17
                   see Section 15.2 }
18
19
       {Datatype Datatype::Create_darray(int size, int rank, int ndims,
20
                   const int array_of_gsizes[], const int array_of_distribs[],
21
                   const int array_of_dargs[], const int array_of_psizes[],
22
                   int order) const(binding deprecated, see Section 15.2) }
23
       {Datatype Datatype::Create_hindexed(int count,
24
                   const int array_of_blocklengths[],
25
                   const Aint array_of_displacements[]) const(binding deprecated, see
26
                   Section 15.2 }
27
28
       {Datatype Datatype::Create_hvector(int count, int blocklength, Aint
29
                   stride) const(binding deprecated, see Section 15.2) }
30
       {Datatype Datatype::Create_indexed_block(int count, int blocklength,
31
                   const int array_of_displacements[]) const(binding deprecated, see
32
                   Section 15.2 }
33
34
       {Datatype Datatype::Create_indexed(int count,
35
                   const int array_of_blocklengths[],
36
                   const int array_of_displacements[]) const(binding deprecated, see
37
                   Section 15.2 }
38
       {Datatype Datatype::Create_resized(const Aint lb, const Aint extent)
39
                   const(binding deprecated, see Section 15.2)
40
41
       {static Datatype Datatype::Create_struct(int count,
42
                   const int array_of_blocklengths[], const Aint
43
                   array_of_displacements[],
44
                   const Datatype array_of_types[])(binding deprecated, see
45
                   Section 15.2 }
46
       {Datatype Datatype::Create_subarray(int ndims,
47
                   const int array_of_sizes[], const int array_of_subsizes[],
48
```

<pre>const int array_of_starts[], int order) const(binding deprecated, see Section 15.2) }</pre>	1 2
<pre>{Datatype Datatype::Create_vector(int count, int blocklength, int stride)</pre>	3 4 5
<pre>{Datatype Datatype::Dup() const(binding deprecated, see Section 15.2) }</pre>	6
<pre>{void Datatype::Free()(binding deprecated, see Section 15.2) }</pre>	7 8
{Aint Get_address(void* location)(binding deprecated, see Section 15.2)}	9
<pre>{void Datatype::Get_contents(int max_integers, int max_addresses,</pre>	10 11 12 13 14
<pre>{int Status::Get_elements(const Datatype& datatype) const(binding deprecated,</pre>	15 16 17
<pre>{void Datatype::Get_envelope(int& num_integers, int& num_addresses,</pre>	18 19 20
<pre>{void Datatype::Get_extent(Aint& lb, Aint& extent) const(binding deprecated,</pre>	21 22 23
<pre>{int Datatype::Get_size() const(binding deprecated, see Section 15.2) }</pre>	23 24
<pre>{void Datatype::Get_true_extent(Aint& true_lb, Aint& true_extent)</pre>	25 26 27
<pre>{void Datatype::Pack(const void* inbuf, int incount, void *outbuf,</pre>	28 29 30
<pre>{void Datatype::Pack_external(const char* datarep, const void* inbuf,</pre>	31 32 33 34
<pre>{Aint Datatype::Pack_external_size(const char* datarep, int incount)</pre>	35 36
<pre>{int Datatype::Pack_size(int incount, const Comm& comm) const(binding</pre>	37 38 39
<pre>{void Datatype::Unpack(const void* inbuf, int insize, void *outbuf,</pre>	40 41 42
<pre>{void Datatype::Unpack_external(const char* datarep, const void* inbuf,</pre>	43 44 45 46
	47 48

};

ANNEX A. LANGUAGE BINDINGS SUMMARY

1	A.5.3 (Collective Communication C++ Bindings
2 3	namespa	ce MPI {
4 5 6 7 8	{void	<pre>Comm::Allgather(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype) const = 0(binding deprecated, see Section 15.2) }</pre>
9 10 11 12	{void	<pre>Comm::Allgatherv(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int displs[], const Datatype& recvtype) const = 0(binding deprecated, see Section 15.2) }</pre>
13 14 15 16	{void	<pre>Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,</pre>
17 18 19 20	{void	<pre>Comm::Alltoall(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype) const = 0(binding deprecated, see Section 15.2) }</pre>
21 22 23 24 25 26	{void	<pre>Comm::Alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], const Datatype& recvtype) const = 0(binding deprecated, see Section 15.2) }</pre>
27 28 29 30	{void	<pre>Comm::Alltoallw(const void* sendbuf, const int sendcounts[], const int sdispls[], const Datatype sendtypes[], void* recvbuf, const int recvcounts[], const int rdispls[], const Datatype recvtypes[]) const = 0(binding deprecated, see Section 15.2) }</pre>
31 32	{void	<pre>Comm::Barrier() const = O(binding deprecated, see Section 15.2) }</pre>
33 34	{void	<pre>Comm::Bcast(void* buffer, int count, const Datatype& datatype,</pre>
35 36 37 38	{void	<pre>Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,</pre>
39	{void	<pre>Op::Free()(binding deprecated, see Section 15.2) }</pre>
40 41 42 43 44	{void	<pre>Comm::Gather(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0(binding deprecated, see Section 15.2) }</pre>
45 46 47 48	{void	<pre>Comm::Gatherv(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int displs[], const Datatype& recvtype, int root) const = O(binding deprecated, see Section 15.2) }</pre>

$\{ {\tt Request Comm}:: {\tt Iallgather(const void* sendbuf, int sendcount, const } \}$	1
Datatype& sendtype, void* recvbuf, int recvcount,	2 3
const Datatype& recvtype) const = $0(binding deprecated, see$	4
Section 15.2) }	5
$\{ {\tt Request Comm}:: {\tt Iallgatherv}({\tt const void}* {\tt sendbuf}, {\tt int sendcount}, {\tt const} \}$	6
<pre>Datatype& sendtype, void* recvbuf, const int recvcounts[],</pre>	7
<pre>const int displs[], const Datatype& recvtype) const = 0(binding</pre>	8
deprecated, see Section 15.2) }	9
{Request Comm::Iallreduce(const void* sendbuf, void* recvbuf, int count,	10
const Datatype& datatype, const Op& op) const = $0(binding$	11
deprecated, see Section 15.2) }	12
{Request Comm::Ialltoall(const void* sendbuf, int sendcount, const	13
Datatype& sendtype, void* recvbuf, int recvcount,	14
const Datatype& recvtype) const = 0(binding deprecated, see	15 16
Section 15.2 }	17
	18
{Request Comm::Ialltoallv(const void* sendbuf, const int sendcounts[],	19
<pre>const int sdispls[], const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int rdispls[],</pre>	20
const Datatype& recvtype) const = 0 (binding deprecated, see	21
Section 15.2) }	22
	23
{Request Comm::Ialltoallw(const void* sendbuf, const int sendcounts[],	24
<pre>const int sdispls[], const Datatype sendtypes[], void* </pre>	25
<pre>recvbuf, const int recvcounts[], const int rdispls[], const Datatype recvtypes[]) const = 0(binding deprecated, see</pre>	26
Section 15.2) }	27
	28
<pre>{Request Comm::Ibarrier() const = O(binding deprecated, see Section 15.2) }</pre>	29 30
{Request Comm::Ibcast(void* buffer, int count, const Datatype& datatype,	31
<pre>int root) const = O(binding deprecated, see Section 15.2) }</pre>	32
	33
<pre>{Request Intracomm::Iexscan(const void* sendbuf, void* recvbuf, int</pre>	34
deprecated, see Section 15.2) }	35
ueprecurcu, see Section 10.2)	36
{Request Comm::Igather(const void* sendbuf, int sendcount, const	37
Datatype& sendtype, void* recvbuf, int recvcount,	38
const Datatype& recvtype, int root) const = 0(binding deprecated,	39
see Section 15.2) }	40
{Request Comm:::Igatherv(const void* sendbuf, int sendcount, const	41
<pre>Datatype& sendtype, void* recvbuf, const int recvcounts[],</pre>	42
<pre>const int displs[], const Datatype& recvtype, int root)</pre>	43 44
<pre>const = O(binding deprecated, see Section 15.2) }</pre>	44
<pre>{void Op::Init(User_function* [function]user_fn, bool commute)(binding</pre>	46 ticket252-W
deprecated, see Section 15.2)	47
· / / J	48

1 2 3	<pre>{Request Comm::Ireduce(const void* sendbuf, void* recvbuf, int count,</pre>
4 5 6 7	<pre>{Request Comm::Ireduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>
8 9 10	<pre>{Request Comm::Ireduce_scatter(const void* sendbuf, void* recvbuf,</pre>
11 12 13 14	<pre>{Request Intracomm::Iscan(const void* sendbuf, void* recvbuf, int count,</pre>
15 16 17 18 19	<pre>{Request Comm::Iscatter(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount,</pre>
20 21 22 23	<pre>{Request Comm::Iscatterv(const void* sendbuf, const int sendcounts[],</pre>
24	<pre>{bool Op::Is_commutative() const(binding deprecated, see Section 15.2) }</pre>
25 26 27 28	<pre>{void Comm::Reduce(const void* sendbuf, void* recvbuf, int count,</pre>
29 30 31 32	<pre>{void Op::Reduce_local(const void* inbuf, void* inoutbuf, int count,</pre>
33 34 35	<pre>{void Comm::Reduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>
36 37 38 39	<pre>{void Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,</pre>
40 41 42	<pre>{void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,</pre>
43 44 45 46 47 48	<pre>{void Comm::Scatter(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0(binding deprecated, see Section 15.2) }</pre>

$\{void Comm:$:Scatterv(const void* sendbuf, const int sendcounts[],	1
	<pre>const int displs[], const Datatype& sendtype, void* recvbuf,</pre>	2
	int recvcount, const Datatype& recvtype, int root)	3
	<pre>const = 0(binding deprecated, see Section 15.2) }</pre>	4 5
		6
};		7
		8
A.5.4 Groups	, Contexts, Communicators, and Caching C++ Bindings	9
namespace MPI	. {	10 11
${Comm & Comm}$:::Clone() const = O(binding deprecated, see Section 15.2) }	12
$\{\texttt{Cartcomm}\&$	<pre>Cartcomm::Clone() const(binding deprecated, see Section 15.2) }</pre>	13 14
{Distgraphc	<pre>comm& Distgraphcomm::Clone() const(binding deprecated, see Section 15.2) }</pre>	15 16
$\{{\tt Graphcomm}\&$	Graphcomm::Clone() const(binding deprecated, see Section 15.2) }	17 18
$\{\texttt{Intercomm}\&$: Intercomm::Clone() const(binding deprecated, see Section 15.2) }	19
$\{\texttt{Intracomm}\&$: Intracomm::Clone() const(binding deprecated, see Section 15.2) }	20 21
$\{ static int $	Comm::Compare(const Comm& comm1, const Comm& comm2)(binding deprecated, see Section 15.2) }	22 23
$\{$ static int	Group::Compare(const Group& group1, const Group& group2)(binding deprecated, see Section 15.2) }	24 25 26
$\{\texttt{Intercomm}$	<pre>Intercomm::Create(const Group& group) const(binding deprecated, see Section 15.2) }</pre>	27 28
$\{ \texttt{Intracomm}$	<pre>Intracomm::Create(const Group& group) const(binding deprecated, see Section 15.2) }</pre>	29 30 31
$\{\texttt{Intercomm}$	<pre>Intracomm::Create_intercomm(int local_leader, const</pre>	32
	<pre>Comm& peer_comm, int remote_leader, int tag) const(binding deprecated, see Section 15.2) }</pre>	33 34
$\{ \texttt{static int} \}$	Comm::Create_keyval(Comm::Copy_attr_function* comm_copy_attr_fn,	35 36
	Comm::Delete_attr_function* comm_delete_attr_fn,	37 38
	<pre>void* extra_state)(binding deprecated, see Section 15.2) }</pre>	39
{static int	Datatype::Create_keyval(Datatype::Copy_attr_function*	40
	<pre>type_copy_attr_fn, Datatype::Delete_attr_function*</pre>	41
	<pre>type_delete_attr_fn, void* extra_state)(binding deprecated, see Section 15.2) }</pre>	42 43
{static int	Win::Create_keyval(Win::Copy_attr_function* win_copy_attr_fn,	44
,	Win::Delete_attr_function* win_delete_attr_fn,	45 46
	<pre>void* extra_state)(binding deprecated, see Section 15.2) }</pre>	40 47
$\{void Comm:$	<pre>:Delete_attr(int comm_keyval)(binding deprecated, see Section 15.2) }</pre>	48

1 2	<pre>{void Datatype::Delete_attr(int type_keyval)(binding deprecated, see Section 15.2) }</pre>
3 4	<pre>{void Win::Delete_attr(int win_keyval)(binding deprecated, see Section 15.2) }</pre>
5 6	<pre>{static Group Group::Difference(const Group& group1,</pre>
7 8	{Cartcomm Cartcomm::Dup() const(binding deprecated, see Section 15.2)}
9 10	{Distgraphcomm Distgraphcomm::Dup() const(binding deprecated, see Section 15.2)}
11	{Graphcomm Graphcomm::Dup() const(binding deprecated, see Section 15.2)}
12 13	{Intercomm Intercomm::Dup() const(binding deprecated, see Section 15.2) }
14	{Intracomm Intracomm::Dup() const(binding deprecated, see Section 15.2)}
15 16 17	<pre>{Group Group::Excl(int n, const int ranks[]) const(binding deprecated, see Section 15.2) }</pre>
18 19	<pre>{static void Comm::Free_keyval(int& comm_keyval)(binding deprecated, see Section 15.2) }</pre>
20 21 22	<pre>{static void Datatype::Free_keyval(int& type_keyval)(binding deprecated, see Section 15.2) }</pre>
23 24	<pre>{static void Win::Free_keyval(int& win_keyval)(binding deprecated, see Section 15.2) }</pre>
25 26	<pre>{void Comm::Free()(binding deprecated, see Section 15.2) }</pre>
27	<pre>{void Group::Free()(binding deprecated, see Section 15.2) }</pre>
28 29 30	<pre>{bool Comm::Get_attr(int comm_keyval, void* attribute_val) const(binding</pre>
31 32	<pre>{bool Datatype::Get_attr(int type_keyval, void* attribute_val)</pre>
33 34 35	<pre>{bool Win::Get_attr(int win_keyval, void* attribute_val) const(binding</pre>
36	{Group Comm::Get_group() const(binding deprecated, see Section 15.2) }
37 38 39	<pre>{void Comm::Get_name(char* comm_name, int& resultlen) const(binding</pre>
40 41	<pre>{void Datatype::Get_name(char* type_name, int& resultlen) const(binding</pre>
42 43 44	<pre>{void Win::Get_name(char* win_name, int& resultlen) const(binding deprecated,</pre>
45	<pre>{int Comm::Get_rank() const(binding deprecated, see Section 15.2) }</pre>
46 47 48	<pre>{int Group::Get_rank() const(binding deprecated, see Section 15.2) }</pre>

<pre>{Group Intercomm::Get_remote_group() const(binding deprecated, see Section 15.2) }</pre>	$\frac{1}{2}$
<pre>{int Intercomm::Get_remote_size() const(binding deprecated, see Section 15.2) }</pre>	$\frac{3}{4}$
<pre>{int Comm::Get_size() const(binding deprecated, see Section 15.2) }</pre>	5
<pre>{int Group::Get_size() const(binding deprecated, see Section 15.2) }</pre>	6 7
<pre>{Group Group::Incl(int n, const int ranks[]) const(binding deprecated, see Section 15.2) }</pre>	8 9 10
<pre>{static Group Group::Intersect(const Group& group1,</pre>	10 11 12
<pre>{bool Comm::Is_inter() const(binding deprecated, see Section 15.2) }</pre>	13 14
<pre>{Intracomm Intercomm::Merge(bool high) const(binding deprecated, see Section 15.2) }</pre>	14 15 16
<pre>{Group Group::Range_excl(int n, const int ranges[][3]) const(binding</pre>	17 18 19
<pre>{Group Group::Range_incl(int n, const int ranges[][3]) const(binding</pre>	20 21
<pre>{void Comm::Set_attr(int comm_keyval, const void* attribute_val)</pre>	22 23 24
<pre>{void Datatype::Set_attr(int type_keyval, const void*</pre>	24 25 26
<pre>{void Win::Set_attr(int win_keyval, const void* attribute_val)(binding</pre>	27 28 29
<pre>{void Comm::Set_name(const char* comm_name)(binding deprecated, see Section 15.2) }</pre>	30 31
<pre>{void Datatype::Set_name(const char* type_name)(binding deprecated, see Section 15.2) }</pre>	32 33 34
<pre>{void Win::Set_name(const char* win_name)(binding deprecated, see Section 15.2) }</pre>	35
<pre>{Intercomm Intercomm::Split(int color, int key) const(binding deprecated, see Section 15.2) }</pre>	36 37 38
<pre>{Intracomm Intracomm::Split(int color, int key) const(binding deprecated, see Section 15.2) }</pre>	39 40
<pre>{static void Group::Translate_ranks (const Group& group1, int n,</pre>	41 42 43 44
<pre>{static Group Group::Union(const Group& group1,</pre>	45 46 47 48

1 2	};				
3	A.5.5 Proces	A.5.5 Process Topologies C++ Bindings			
4 5	namespace MP	namespace MPI {			
6 7 8	{void Compu	<pre>ute_dims(int nnodes, int ndims, int dims[])(binding deprecated, see Section 15.2) }</pre>			
9 10 11	{Cartcomm]	<pre>Intracomm::Create_cart(int ndims, const int dims[], const bool periods[], bool reorder) const(binding deprecated, see Section 15.2) }</pre>			
12 13 14 15	$\{{\tt Graphcomm}$	<pre>Intracomm::Create_graph(int nnodes, const int index[], const int edges[], bool reorder) const(binding deprecated, see Section 15.2) }</pre>			
16 17 18 19 20	{Distgraph	<pre>comm Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], const int sourceweights[], int outdegree, const int destinations[], const int destweights[], const Info& info, bool reorder) const(binding deprecated, see Section 15.2) }</pre>			
21 22 23 24 25	{Distgraph	<pre>comm Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], int outdegree, const int destinations[], const Info& info, bool reorder) const(binding deprecated, see Section 15.2) }</pre>			
26 27 28 29 30	{Distgraph	<pre>comm Intracomm::Dist_graph_create(int n, const int sources[], const int degrees[], const int destinations[], const int weights[], const Info& info, bool reorder) const(binding deprecated, see Section 15.2) }</pre>			
31 32 33 34	{Distgraph	<pre>comm Intracomm::Dist_graph_create(int n, const int sources[], const int degrees[], const int destinations[], const Info& info, bool reorder) const(binding deprecated, see Section 15.2) }</pre>			
35 36 37	$\{$ int Cartco	<pre>pmm::Get_cart_rank(const int coords[]) const(binding deprecated, see Section 15.2) }</pre>			
38 39	$\{void Carto$	<pre>comm::Get_coords(int rank, int maxdims, int coords[]) const(binding deprecated, see Section 15.2) }</pre>			
40 41	{int Cartco	<pre>pmm::Get_dim() const(binding deprecated, see Section 15.2) }</pre>			
42 43	$\{void Graph\}$	<pre>ncomm::Get_dims(int nnodes[], int nedges[]) const(binding deprecated, see Section 15.2) }</pre>			
44 45 46 47 48	{void Distg	<pre>graphcomm::Get_dist_neighbors_count(int rank, int indegree[], int outdegree[], bool& weighted) const(binding deprecated, see Section 15.2) }</pre>			

<pre>int sourceweights[], int maxoutdegree, int destinations[], int destweights[])(binding deprecated, see Section 15.2) }</pre>	2 3
<pre>{int Graphcomm::Get_neighbors_count(int rank) const(binding deprecated, see Section 15.2) }</pre>	4 5 6
<pre>{void Graphcomm::Get_neighbors(int rank, int maxneighbors, int neighbors[]) const(binding deprecated, see Section 15.2) }</pre>	7 8
<pre>{void Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],</pre>	9 10 11
<pre>{void Graphcomm::Get_topo(int maxindex, int maxedges, int index[],</pre>	12 13
<pre>{int Comm::Get_topology() const(binding deprecated, see Section 15.2) }</pre>	14 15
<pre>{int Cartcomm::Map(int ndims, const int dims[], const bool periods[])</pre>	16 17 18
<pre>{int Graphcomm::Map(int nnodes, const int index[], const int edges[])</pre>	19 20
<pre>{void Cartcomm::Shift(int direction, int disp, int& rank_source,</pre>	21 22 23
<pre>{Cartcomm Cartcomm::Sub(const bool remain_dims[]) const(binding deprecated,</pre>	24 25
};	26 27 28
A.5.6 MPI Environmental Management C++ Bindings	29
namespace MPI {	30 31
<pre>{void Comm::Abort(int errorcode)(binding deprecated, see Section 15.2) }</pre>	32
<pre>{int Add_error_class()(binding deprecated, see Section 15.2) }</pre>	33 34
{int Add_error_code(int errorclass)(binding deprecated, see Section 15.2)}	35 36
<pre>{void Add_error_string(int errorcode, const char* string)(binding deprecate see Section 15.2) }</pre>	
<pre>{void* Alloc_mem(Aint size, const Info& info)(binding deprecated, see Section 15.2) }</pre>	39 40 41
<pre>{void Comm::Call_errhandler(int errorcode) const(binding deprecated, see Section 15.2) }</pre>	42 43
<pre>{void File::Call_errhandler(int errorcode) const(binding deprecated, see Section 15.2) }</pre>	44 45 46
<pre>{void Win::Call_errhandler(int errorcode) const(binding deprecated, see Section 15.2) }</pre>	47 48

ANNEX A. LANGUAGE BINDINGS SUMMARY

1 icket252-W. 2	<pre>{static Errhandler Comm::Create_errhandler(Comm::Errhandler_function* [function]comm_errhandler_fn)(binding deprecated, see Section 15.2) }</pre>
icket252-W. $\frac{4}{5}$	<pre>{static Errhandler File::Create_errhandler(File::Errhandler_function* [function]file_errhandler_fn)(binding deprecated, see Section 15.2) }</pre>
⁶ icket252-W. ⁷	<pre>{static Errhandler Win::Create_errhandler(Win::Errhandler_function* [function]win_errhandler_fn)(binding deprecated, see Section 15.2) }</pre>
8	<pre>{void Finalize()(binding deprecated, see Section 15.2) }</pre>
9 10	
11	{void Free_mem(void *base) (binding deprecated, see Section 15.2) }
12	{void Errhandler::Free()(binding deprecated, see Section 15.2)}
13	{Errhandler Comm::Get_errhandler() const(binding deprecated, see Section 15.2)}
14 15	{Errhandler File::Get_errhandler() const(binding deprecated, see Section 15.2)}
16	{Errhandler Win::Get_errhandler() const(binding deprecated, see Section 15.2) }
17 18	<pre>{int Get_error_class(int errorcode)(binding deprecated, see Section 15.2) }</pre>
19 20	<pre>{void Get_error_string(int errorcode, char* name, int& resultlen)(binding</pre>
21 22 23	<pre>{void Get_processor_name(char* name, int& resultlen)(binding deprecated, see Section 15.2) }</pre>
24 25 26	<pre>{void Get_version(int& version, int& subversion)(binding deprecated, see Section 15.2) }</pre>
20	<pre>{void Init(int& argc, char**& argv)(binding deprecated, see Section 15.2) }</pre>
28 29	<pre>{void Init()(binding deprecated, see Section 15.2) }</pre>
30	{bool Is_finalized()(binding deprecated, see Section 15.2)}
31 32	<pre>{bool Is_initialized()(binding deprecated, see Section 15.2) }</pre>
33 34	{void Comm::Set_errhandler(const Errhandler& errhandler)(binding deprecated,
35 36	<pre>see Section 15.2) } {void File::Set_errhandler(const Errhandler& errhandler)(binding deprecated,</pre>
37 38	{void Win::Set_errhandler(const Errhandler& errhandler)(binding deprecated,

{double Wtick()(binding deprecated, see Section 15.2)}

see Section 15.2) }

{double Wtime()(binding deprecated, see Section 15.2) }

A.5.7 The Info Object C++ Bindings

 48 namespace MPI {

39

40

41 42

43

<pre>{static Info Info::Create()(binding deprecated, see Section 15.2) }</pre>	1
<pre>{void Info::Delete(const char* key)(binding deprecated, see Section 15.2) }</pre>	2 3
{Info Info::Dup() const(binding deprecated, see Section 15.2) }	4
<pre>{void Info::Free()(binding deprecated, see Section 15.2)}</pre>	5 6
{bool Info::Get(const char* key, int valuelen, char* value) const(binding	7
deprecated, see Section 15.2) }	8
<pre>{int Info::Get_nkeys() const(binding deprecated, see Section 15.2) }</pre>	9 10
{void Info::Get_nthkey(int n, char* key) const(binding deprecated, see	11
Section 15.2 }	12 13
{bool Info::Get_valuelen(const char* key, int& valuelen) $const(binding$	14
deprecated, see Section 15.2 }	15
{void Info::Set(const char* key, const char* value)(binding deprecated, see	16 17
Section 15.2 }	18
};	19 20
	20
A.5.8 Process Creation and Management C++ Bindings	22
namespace MPI {	23 24
{Intercomm Intracomm::Accept(const char* port_name, const Info& info,	25
<pre>int root) const(binding deprecated, see Section 15.2) }</pre>	26 27
<pre>{void Close_port(const char* port_name)(binding deprecated, see Section 15.2) }</pre>	28
{Intercomm Intracomm::Connect(const char* port_name, const Info& info,	29
<pre>int root) const(binding deprecated, see Section 15.2) }</pre>	30 31
<pre>{void Comm::Disconnect()(binding deprecated, see Section 15.2) }</pre>	32
{static Intercomm Comm::Get_parent()(binding deprecated, see Section 15.2)}	33 34
{static Intercomm Comm::Join(const int fd)(binding deprecated, see Section 15.2)	
}	36
<pre>{void Lookup_name(const char* service_name, const Info& info,</pre>	37 38
<pre>char* port_name)(binding deprecated, see Section 15.2) }</pre>	39
<pre>{void Open_port(const Info& info, char* port_name)(binding deprecated, see Section 15.2) }</pre>	40 41
	42
<pre>{void Publish_name(const char* service_name, const Info& info,</pre>	43
{Intercomm Intracomm::Spawn(const char* command, const char* argv[],	44 45
int maxprocs, const Info& info, int root) const (binding	46
deprecated, see Section 15.2) }	47
	48

```
1
       {Intercomm Intracomm::Spawn(const char* command, const char* argv[],
2
                    int maxprocs, const Info& info, int root,
3
                    int array_of_errcodes[]) const(binding deprecated, see Section 15.2)
4
                    }
5
       {Intercomm Intracomm::Spawn_multiple(int count,
6
                    const char* array_of_commands[], const char** array_of_argv[],
7
                    const int array_of_maxprocs[], const Info array_of_info[],
8
                    int root, int array_of_errcodes[]) (binding deprecated, see
9
                    Section 15.2 }
10
11
       {Intercomm Intracomm::Spawn_multiple(int count,
12
                    const char* array_of_commands[], const char** array_of_argv[],
13
                    const int array_of_maxprocs[], const Info array_of_info[],
14
                    int root) (binding deprecated, see Section 15.2) }
15
       {void Unpublish_name(const char* service_name, const Info& info,
16
                    const char* port_name) (binding deprecated, see Section 15.2) }
17
18
19
     };
20
21
     A.5.9 One-Sided Communications C++ Bindings
22
     namespace MPI {
23
24
       {void Win::Accumulate(const void* origin_addr, int origin_count, const
25
                    Datatype& origin_datatype, int target_rank, Aint target_disp,
26
                    int target_count, const Datatype& target_datatype, const Op&
27
                    op) const (binding deprecated, see Section 15.2) }
28
29
       {void Win::Complete() const(binding deprecated, see Section 15.2) }
30
       {static Win Win::Create(const void* base, Aint size, int disp_unit, const
31
                    Info& info, const Intracomm& comm) (binding deprecated, see
32
                    Section 15.2 }
33
34
       {void Win::Fence(int assert) const(binding deprecated, see Section 15.2) }
35
       {void Win::Free()(binding deprecated, see Section 15.2) }
36
37
       {Group Win::Get_group() const(binding deprecated, see Section 15.2) }
38
       {void Win::Get(void *origin_addr, int origin_count,
39
                    const Datatype& origin_datatype, int target_rank,
40
                    Aint target_disp, int target_count,
41
                    const Datatype& target_datatype) const(binding deprecated, see
42
                    Section 15.2 }
43
44
       {void Win::Lock(int lock_type, int rank, int assert) const(binding
45
                    deprecated, see Section 15.2 }
46
       {void Win::Post(const Group& group, int assert) const(binding deprecated, see
47
                    Section 15.2 }
48
```

```
1
  {void Win::Put(const void* origin_addr, int origin_count,
                                                                                         2
               const Datatype& origin_datatype, int target_rank,
                                                                                         3
              Aint target_disp, int target_count,
               const Datatype& target_datatype) const(binding deprecated, see
                                                                                         4
               Section 15.2 }
                                                                                         5
                                                                                         6
  {void Win::Start(const Group& group, int assert) const/binding deprecated.
                                                                                         7
               see Section 15.2 }
                                                                                         8
                                                                                         9
  {bool Win::Test() const(binding deprecated, see Section 15.2) }
                                                                                         10
  {void Win::Unlock(int rank) const(binding deprecated, see Section 15.2) }
                                                                                         11
                                                                                         12
  {void Win::Wait() const(binding deprecated, see Section 15.2) }
                                                                                         13
                                                                                         14
};
                                                                                         15
                                                                                         16
A.5.10 External Interfaces C++ Bindings
                                                                                         17
                                                                                         18
namespace MPI {
                                                                                         19
                                                                                         20
  {void Grequest::Complete()(binding deprecated, see Section 15.2) }
                                                                                         21
  {int Init_thread(int& argc, char**& argv, int required)(binding deprecated,
                                                                                         22
               see Section 15.2 }
                                                                                         23
                                                                                         24
  {int Init_thread(int required)(binding deprecated, see Section 15.2) }
                                                                                         25
  {bool Is_thread_main() (binding deprecated, see Section 15.2) }
                                                                                         26
                                                                                         27
  {int Query_thread() (binding deprecated, see Section 15.2) }
                                                                                         28
  {void Status::Set_cancelled(bool flag) (binding deprecated, see Section 15.2) }
                                                                                         29
                                                                                         30
  {void Status::Set_elements(const Datatype& datatype, int count)(binding
                                                                                         31
               deprecated, see Section 15.2 }
                                                                                         32
  {static Grequest Grequest::Start(const Grequest::Query_function*
                                                                                         33
              query_fn, const Grequest::Free_function* free_fn,
                                                                                         34
              const Grequest::Cancel_function* cancel_fn,
                                                                                         35
              void *extra_state) (binding deprecated, see Section 15.2) }
                                                                                         36
                                                                                         37
                                                                                         38
};
                                                                                         39
                                                                                         40
A.5.11 I/O C++ Bindings
                                                                                         41
namespace MPI {
                                                                                         42
                                                                                         43
  {void File::Close() (binding deprecated, see Section 15.2) }
                                                                                         44
                                                                                         45
  {static void File::Delete(const char* filename, const Info& info)(binding
                                                                                         46
               deprecated, see Section 15.2 }
                                                                                         47
  {int File::Get_amode() const(binding deprecated, see Section 15.2) }
                                                                                         48
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

1	<pre>{bool File::Get_atomicity() const(binding deprecated, see Section 15.2) }</pre>
2 3 4	<pre>{Offset File::Get_byte_offset(const Offset disp) const(binding deprecated,</pre>
5	{Group File::Get_group() const(binding deprecated, see Section 15.2) }
6 7	<pre>{Info File::Get_info() const(binding deprecated, see Section 15.2) }</pre>
8 9	{Offset File::Get_position() const(binding deprecated, see Section 15.2) }
9 10	<pre>{Offset File::Get_position_shared() const(binding deprecated, see Section 15.2) }</pre>
11 12	<pre>{Offset File::Get_size() const(binding deprecated, see Section 15.2) }</pre>
12 13 14	<pre>{Aint File::Get_type_extent(const Datatype& datatype) const(binding</pre>
15 16 17	<pre>{void File::Get_view(Offset& disp, Datatype& etype, Datatype& filetype,</pre>
18 19	<pre>{Request File::Iread_at(Offset offset, void* buf, int count,</pre>
20 21 22	<pre>{Request File::Iread_shared(void* buf, int count,</pre>
23 24	<pre>{Request File::Iread(void* buf, int count,</pre>
25 26 27	<pre>{Request File::Iwrite_at(Offset offset, const void* buf, int count,</pre>
28 29	<pre>{Request File::Iwrite(const void* buf, int count,</pre>
30 31 32	<pre>{Request File::Iwrite_shared(const void* buf, int count,</pre>
33 34	<pre>{static File File::Open(const Intracomm& comm, const char* filename,</pre>
35 36	{void File::Preallocate(Offset size)(binding deprecated, see Section 15.2)}
37 38	<pre>{void File::Read_all_begin(void* buf, int count,</pre>
39 40 41	<pre>{void File::Read_all_end(void* buf, Status& status)(binding deprecated, see Section 15.2) }</pre>
42	<pre>{void File::Read_all_end(void* buf)(binding deprecated, see Section 15.2) }</pre>
43 44 45	<pre>{void File::Read_all(void* buf, int count, const Datatype& datatype, Status& status)(binding deprecated, see Section 15.2) }</pre>
46 47 48	<pre>{void File::Read_all(void* buf, int count,</pre>

{void	The mead_at_att_begin(bilset bilset, void* but, int count,	1 2
{void	<pre>File::Read_at_all_end(void* buf, Status& status)(binding deprecated,</pre>	3 4 5
{void	<pre>File::Read_at_all_end(void* buf)(binding deprecated, see Section 15.2) }</pre>	6
{void	<pre>File::Read_at_all(Offset offset, void* buf, int count,</pre>	7 8 9 10
{void	const Datatype& datatype) (binding deprecated, see Section 15.2) }	11 12 13
{void	File::Read_at(Offset offset, void* buf, int count, const Datatype& datatype, Status& status)(binding deprecated, see Section 15.2)]	14 15 16
{void	const Datatype& datatype) (binding deprecated, see Section 15.2) }	17 18 19
{void	File::Read_ordered_begin(void* buf, int count,	20 21
{void	<pre>File::Read_ordered_end(void* buf, Status& status)(binding deprecated,</pre>	22 23 24
{void	<pre>File::Read_ordered_end(void* buf)(binding deprecated, see Section 15.2) }</pre>	25
{void	<pre>File::Read_ordered(void* buf, int count, const Datatype& datatype,</pre>	26 27 28
{void	<pre>const Datatype& datatype)(binding deprecated, see Section 15.2) }</pre>	29 30
{void	File::Read_shared(void* buf, int count, const Datatype& datatype, Status& status)(binding deprecated, see Section 15.2) }	31 32 33
{void	<pre>const Datatype& datatype)(binding deprecated, see Section 15.2) }</pre>	34 35
{void	File::Read(void* buf, int count, const Datatype& datatype, Status& status)(binding deprecated, see Section 15.2) }	36 37 38
{void	deprecated, see Section 15.2) }	39 10
{void	<pre>Register_datarep(const char* datarep, Datarep_conversion_function* read_conversion_fn, Datarep_conversion_function* write_conversion_fn, Datarep_extent_function* dtype_file_extent_fn, void* extra_state)(binding deprecated, see Section 15.2) }</pre>	 41 42 43 44 45 46 47
	4	18

1 2	{void	<pre>File::Seek(Offset offset, int whence)(binding deprecated, see Section 15.2) }</pre>
3 4 5	{void	<pre>File::Seek_shared(Offset offset, int whence)(binding deprecated, see Section 15.2) }</pre>
6	{void	<pre>File::Set_atomicity(bool flag)(binding deprecated, see Section 15.2) }</pre>
7 8	{void	<pre>File::Set_info(const Info& info)(binding deprecated, see Section 15.2) }</pre>
9 10	{void	<pre>File::Set_size(Offset size)(binding deprecated, see Section 15.2) }</pre>
10 11 12 13	{void	<pre>File::Set_view(Offset disp, const Datatype& etype,</pre>
14 15	{void	<pre>File::Sync()(binding deprecated, see Section 15.2) }</pre>
16 17	{void	<pre>File::Write_all_begin(const void* buf, int count,</pre>
18 19 20 21	{void	<pre>File::Write_all(const void* buf, int count,</pre>
22 23	{void	<pre>File::Write_all(const void* buf, int count,</pre>
24 25 26	{void	<pre>File::Write_all_end(const void* buf, Status& status)(binding</pre>
27 28	{void	<pre>File::Write_all_end(const void* buf)(binding deprecated, see Section 15.2) }</pre>
29 30 31	{void	<pre>File::Write_at_all_begin(Offset offset, const void* buf, int count,</pre>
32 33	{void	<pre>File::Write_at_all_end(const void* buf, Status& status)(binding</pre>
34 35 36	{void	<pre>File::Write_at_all_end(const void* buf)(binding deprecated, see Section 15.2) }</pre>
37 38 39	{void	<pre>File::Write_at_all(Offset offset, const void* buf, int count,</pre>
40 41 42	{void	<pre>File::Write_at_all(Offset offset, const void* buf, int count,</pre>
43 44 45	{void	<pre>File::Write_at(Offset offset, const void* buf, int count,</pre>
46 47 48	{void	<pre>File::Write_at(Offset offset, const void* buf, int count,</pre>

	e(const void* buf, int count, const Datatype& datatype, us& status)(binding deprecated, see Section 15.2) }	$\frac{1}{2}$
	e(const void* buf, int count, t Datatype& datatype) <i>(binding deprecated, see Section 15.2)</i> }	3 4 5
{void File::Writ	<pre>se_ordered_begin(const void* buf, int count, t Datatype& datatype)(binding deprecated, see Section 15.2) }</pre>	5 6 7
{void File::Writ cons	<pre>se_ordered(const void* buf, int count, t Datatype& datatype, Status& status)(binding deprecated, see on 15.2) }</pre>	8 9 10 11
	<pre>se_ordered(const void* buf, int count, t Datatype& datatype)(binding deprecated, see Section 15.2) }</pre>	12 13
	<pre>se_ordered_end(const void* buf, Status& status)(binding ecated, see Section 15.2) }</pre>	14 15 16
	<pre>se_ordered_end(const void* buf)(binding deprecated, see on 15.2) }</pre>	17 18
cons	<pre>se_shared(const void* buf, int count, t Datatype& datatype, Status& status)(binding deprecated, see on 15.2) }</pre>	19 20 21 22
	<pre>se_shared(const void* buf, int count, t Datatype& datatype)(binding deprecated, see Section 15.2) }</pre>	23 24 25
};		26 27
A.5.12 Language B	indings C++ Bindings	28 29
namespace MPI {		30
	<pre>e Datatype::Create_f90_complex(int p, int r)(binding ecated, see Section 15.2) }</pre>	31 32 33
	<pre>Datatype::Create_f90_integer(int r)(binding deprecated, see on 15.2) }</pre>	34 35
· · ·	<pre>e Datatype::Create_f90_real(int p, int r)(binding deprecated, lection 15.2) }</pre>	36 37 38
Exception::Exception:	<pre>tion(int error_code)</pre>	39
{int Exception::	<pre>Get_error_class() const(binding deprecated, see Section 15.2) }</pre>	40 41
{int Exception::	<pre>Get_error_code() const(binding deprecated, see Section 15.2) }</pre>	42
	<pre>ception::Get_error_string() const(binding deprecated, see on 15.2) }</pre>	43 44 45
	<pre>Datatype::Match_size(int typeclass, int size)(binding scated, see Section 15.2) }</pre>	46 47 48

```
1
               };
         \mathbf{2}
         3
               A.5.13 Profiling Interface C++ Bindings
         4
               namespace MPI {
         5
         6
                  {void Pcontrol(const int level, ...) (binding deprecated, see Section 15.2) }
         7
          8
               };
ticket11. 9
               [C++ Deprecated Functions section]
         10
         11
               A.5.14 C++ Bindings on all MPI Classes
         12
         13
               The C++ language requires all classes to have four special functions: a default constructor,
         14
               a copy constructor, a destructor, and an assignment operator. The bindings for these func-
         15
               tions are listed below; their semantics are discussed in Section 16.1.5. The two constructors
         16
               are not virtual. The bindings prototype functions are using the type (CLASS) rather than
         17
               listing each function for every MPI class. The token (CLASS) can be replaced with valid MPI-
         18
               2 class names, such as Group, Datatype, etc., except when noted. In addition, bindings are
         19
               provided for comparison and inter-language operability from Sections 16.1.5 and 16.1.9.
         20
         21
               A.5.15 Construction / Destruction
         22
         23
               namespace MPI {
         ^{24}
                  \langle CLASS \rangle : : \langle CLASS \rangle ()
         25
         26
                  \langle \text{CLASS} \rangle :: \sim \langle \text{CLASS} \rangle ()
         27
         28
               };
         29
         30
               A.5.16 Copy / Assignment
         ^{31}
               namespace MPI {
         32
         33
                  (CLASS)::(CLASS)(const (CLASS)& data)
         34
                  (CLASS)& (CLASS)::operator=(const (CLASS)& data)
         35
         36
               };
         37
         38
               A.5.17 Comparison
         39
         40
               Since Status instances are not handles to underlying MPI objects, the operator==() and
         41
               operator!=() functions are not defined on the Status class.
         42
               namespace MPI {
         43
         44
                  bool (CLASS)::operator==(const (CLASS)& data) const
         45
                  bool (CLASS)::operator!=(const (CLASS)& data) const
         46
         47
         48
               };
```

A.5.18 Inter-language Operability

Since there are no C++ MPI::STATUS_IGNORE and MPI::STATUSES_IGNORE objects, the result of promoting the C or Fortran handles (MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE) to C++ is undefined.

};

Annex B

Change-Log

This annex summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown.

B.1	Changes from Version 2.2 to Version 3.0	20 ²¹ ticket166.
1.	Section 16.1.6 on page 633, and MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 errata: The constant MPI::_LONG_LONG should be MPI::LONG_LONG. TICKET NOT YET PASSED. NEW CHANGE-LOG TEXT.	22 23 24 25 26 ticket171.
2.	Section 13.5.2, Table 13.2 on page 562, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433. This is an MPI-2.2 errata: The MPI_C_BOOL "external32" representation is corrected to a 1-byte size. TICKET PASSED. NEW CHANGE-LOG TEXT.	27 28 29 30 $^{31}_{32}$ ticket192.
3.	Section 7.5.5 on page 315, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3. This is an MPI-2.2 errata: In the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COU the argument rank is removed. TICKET NOT YET PASSED (Had 1st vote). NEW CHANGE-LOG TEXT.	36 37 ticket202.
4.	Annex A.1.1 on page 711, Table "Optional datatypes (Fortran)", and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37-41. This is an MPI-2.2 errata: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 where added to the table. TICKET NOT YET PASSED. NEW CHANGE-LOG TEXT.	 38 39 40 41 42 43 44 ticket274.
5.	Sections 3.8.2, 3.8.3, 16.3.4, A.1.1 on pages 75, 76, 695, 711. Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism	 45 ticket38. 46 47 48

 18 ticket0.

	1	to receive the specific message with the new routine MPI_MRECV regardless of other
	2	intervening probe or receive operations. The opaque object MPI_Message, the null
	3	handle MPI_MESSAGE_NULL, and the conversion functions MPI_Message_c2f and
	4	MPI_Message_f2c are defined.
	5	-
ticket109	•	TICKETS 38+274 PASSED. NEW CHANGE-LOG TEXT.
	0	6. Chapter 5 on page 151 and Section 5.12 on page 209.
		Added nonblocking interfaces to all collective operations.
	8	· ·
ticket140	. 9	TEXT AS PASSED.
	10	7. Section 2.3 on page 10.
	11	
	12	Clarified parameter usage for IN parameters. C bindings are now const-correct where
	13	backward compatiblity is preserved.
ticket125		TICKET AS PASSED. Reference updated from Section 2 to Section 2.3
ticket126		
ticket140		8. Chapter 3 on page 27 until Chapter 16 on page 629.
UCKC0140	• 16	In the C language bindings, the array-arguments' interfaces are modified to consis-
	17	tently to always use use [] instead of *, and the 'const' keyword has been added to
	18	many functions.
1.1.100	19	TICKET PASSED. NEW CHANGE-LOG TEXT.
ticket162	· 20	
	21	9. Section 7.5.8 on page 326.
	22	MPI_CART_MAP can also be used for a zero-dimensional topologies.
		TEXT AS PASSED.
ticket168		
	²⁴ 1	0. Section 6.4.2 on page 252.
	25	Added MPI_COMM_IDUP.
1 1 1 20 4	26	TEXT AS PASSED.
ticket204	• 27	TEAT AS TROOLD.
	28 1	1. Section 2.5.4 on page 15 and Section 8.1.1 on page 349.
	29	Added new routine MPI_GET_LIBRARY_VERSION to query library specific versions,
	30	and the constant MPI_MAX_LIBRARY_VERSION_STRING.
	31	
ticket 219		TICKET PASSED. MODIFIED CHANGE-LOG TEXT.
	32 1	2. Section 6.8 on page 294.
	33 1	Section 6.8 on page 234. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type
	34	
ticket 222	· ³⁵	and window names.
	³⁶ 1	3. Section ?? on page ??.
	37 1	
	38	I ASKED GEORGE TO SET THE MISSING LABEL AT "12.4.3 Initialization"
	39	IT MUST BE DECIDED, WHICH OPTION WE USE ABOUT SAME
	40	required ARGUMENT WHEN CALLING MPI_INIT_THREAD.
ticket328		TICKET NOT YET PASSED. CHANGE-LOG TEXT MUST BE ALSO DEFINED.
ticket256		
5101100200	· 42 1	4. Section 3.8 on page 71 and Section 3.11 on page 89.
	43	The use of MPI_PROC_NULL in probe and matching probe operations was clarified. A
	44	special predefined message MPI_MESSAGE_NO_PROC is defined for the use of matching
	45	probe with MPI_PROC_NULL.
	46	TICKET 256 NOT YET PASSED (Had 1st vote). CHANGE-LOG TEXT AS DEFINED IN TICKET.
	47	TICKET 328 NOT YET PASSED (Had 1st vote). NEW CHANGE-LOG TEXT.
	48	TOALT 525 NOT TET TASSED (Hau 180 VOIC). NEW OHANGE-LOG TEAT.

ticket299. 1 ticket258. $\mathbf{2}$ 15. Section 7.6 on page 328 and Section 7.7 on page 337. 3 The neighborhood collective communication routines are added to support sparse 4 communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, 5MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, 6 MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking 7 variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 8 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 9 MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in 10 MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW are defined as ad-11 dress size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule is added for 12communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT. 13 TICKET PASSED. MODIFIED CHANGE-LOG TEXT. $_{14}$ ticket 265. 16. Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 17, 29, 31, 188, Sections ??, ??, ??, 4.1.11, 1512.3 on pages ??, ??, ??, 121, 501, and Annex A.1.1 on page 711. 16New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, 17 MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their re-18 sults as an MPI_Count value, which is a new type large enough to represent element 19 counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, 20modifies the opaque part of MPI_STATUS so that a call to MPI_GET_ELEMENTS_X re-21turns the provided MPI_Count value (in Fortran, INTEGER (KIND=MPI_COUNT_KIND)). 22 The corresponding predefined datatype is MPI_COUNT. 23 TICKET PASSED. CHANGE-LOG TEXT EXTENDED BY Fortran AND predefined datatype. 24ticket265. 25 17. Sections ??, ??, ??, ?? on pages ??, ??, ??, ??. 26The functions MPI_GET_COUNT, MPI_TYPE_SIZE, and MPI_GET_ELEMENTS are 27now defined to set the count parameter to MPI_UNDEFINED when that parameter 28 would overflow. The function MPI_PACK_SIZE is now defined to set the size param-29 eter to MPI_UNDEFINED when that parameter would overflow. In all other MPI-2.2 30 routines, the type and semantics of the count arguments are kept unchanged, i.e., int 31 or INTEGER. 32 TEXT AS PASSED. 33 ticket266. 34 18. Section 8.7 on page 373. 35 Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE. 36 TICKET PASSED. MODIFIED CHANGE-LOG TEXT. ticket266. 37 19. Section 14.3 on page 590. 38 A new MPI Tool Information Interface is added. 39 TICKET PASSED. MODIFIED CHANGE-LOG TEXT. 40 ticket 284. ⁴¹ ticket300. 20. Chapter 11 on page 421. 42 ticket 270. Substantial revision of the entire One-sided chapter, with new routines for window 43 creation, additional synchronization methods in passive target, new one-sided com-44 munication routines, a new memory model, and other changes. 45 Ticket 270, TEXT AS PASSED. Ticket 284, NO ADDITIONAL CHANGE-LOG 46 ticket271. 47 21. Sections 6.4.2, ??, ??, on pages 252, ??, ??. 48 I ASKED ADAM FOR THE MISSING LABELS of "6.4.4 Communicator Info" and "11.2.3 Window Info"

	1	The new routines MPI_COMM_DUP_INFO, MPI_COMM_SET_INFO,
	2	MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO are added.
	3	The routine MPI_COMM_DUP must also duplicate topology information and info
	4	hints.
ticket279.	5	TICKET NOT YET PASSED (Had 1st vote). NEW CHANGE-LOG TEXT.
01CRC0210.	6	HORET NOT TET TASSED (Had ist vote). New CHANGE-LOG TEXT.
	7	22. Chapter 16.1.1 on page 629.
	8	Added a sentence making the C++ bindings optional.
	9	TEXT AS PASSED.
ticket280.	10	CAUTION: MAY BE OBSOLETE WITH TICKET 281!
	11	22 Caption 4.1.2 on page 02 and Caption 4.1.12 on page 194
	12	23. Section 4.1.2 on page 93 and Section 4.1.13 on page 124.
	13	The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK are added.
	14	
ticket 281.	15	TICKET PASSED. MODIFIED CHANGE-LOG TEXT.
	16	24. Section ?? on page ?? and all other chapters.
	17	THIS LABEL IS ONLY AVAILABLE AFTER TICKET 281 IS INCLUDED.
	18	The C++ bindings are removed from the standard. See MPI-2.2 errata at the begin-
	19	ning of this list for latest changes to the $MPI C++$ binding defined in $MPI-2.2$.
ticket286.	20	TICKET NOT YET PASSED (Had 1st vote). NEW CHANGE-LOG TEXT.
UICKC0200.	21	
	22	25. Section 6.4.2 on page 252.
	23	New communicator construction routine MPI_COMM_CREATE_GROUP, which is in-
	24	voked only by the processes in the group of the new communicator being constructed.
ticket 287.		TICKET PASSED. NEW CHANGE-LOG TEXT.
	26	26. Section 6.4.2 on page 252.
	27	Added MPI_COMM_SPLIT_TYPE function and the communicator split type constant
	28	MPI_COMM_TYPE_SHARED.
ticket294.	29	TICKET PASSED. MODIFIED CHANGE-LOG TEXT.
01011002011		
	32	27. Section 2.5.4 on page 15 and Section 7.5.4 on page 308.
	33	The recommended C implementation value for MPI_UNWEIGHTED was changed from
	34	NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was introduced.
	35	
ticket303.	36	TICKET NOT YET PASSED (Had 1st vote). CHANGE-LOG TEXT AS DEFINED IN TICKET.
	37	28. Section 15.1 on page 619 and Section ?? on page ??.
	38	The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED,
	39	MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB,
	40	MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype
	41	${\sf MPI_Handler_function}, \ {\sf MPI_ERRHANDLER_SET}, \ {\sf MPI_ERRHANDLER_GET}, \ {\rm the} \ {\rm dep}{\rm -}$
	42	recated special data type handles $MPI_LB,$ $MPI_UB,$ and the constants
	43	MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER,
	44	$MPI_COMBINER_STRUCT_INTEGER$ are removed from the standard.
ticket305.	45	TICKET NOT YET PASSED (Had 1st vote). NEW CHANGE-LOG TEXT.
	46	29. Section 6.6.2 on page 271.
	47	The scope of the tag argument in MPI_INTERCOMM_CREATE is shrunk to the use
	48	The scope of the tag argument in the Line Encountry_encenter is smallk to the use

	in this routine. TICKET PASSED. NEW CHANGE-LOG TEXT.	$\frac{1}{2}$ ticket313.
30.	Section 8.7 on page 373 and Section ?? on page ??. I ASKED GEORGE TO SET THE MISSING LABEL AT "12.4.3 Initialization" The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE is clarified. After MPI is initialized, the application can access information about the execution environment by querying the new predefined info object MPI_INFO_GET_ENV. TICKET NOT YET PASSED (Had 1st vote). NEW CHANGE-LOG TEXT. CAUTION: There are three locations of MPI_INFO_KEY. I expect that this is a typo and should	tICKEt515. 3 4 5 6 7 8 9 10
31.	mean MPI_INFO_GET_ENV. Sections 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 188, 562, and 711. New named optional predefined datatypes MPI_QUAD, MPI_C_QUAD_COMPLEX, MPI_FLOAT128, and MPI_C_FLOAT128_COMPLEX for the C types _Quad, float128, _Quad _Complex, andfloat128 _Complex. TICKET NOT YET PASSED. CHANGE-LOG TEXT AS ON TICKET.	¹¹ ticket318. ¹² ¹³ ¹⁴ ¹⁵ ¹⁶ ₁₇ ticket322.
32.	Section 6.7.2 on page 278. Section 6.7.2 on page 226. It was clarified that in Fortran, the flag values of a comm_copy_attr_fn callback and of MPI_COMM_NULL_COPY_FN and MPI_COMM_DUP_FN are .FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL. TICKET PASSED. MODIFIED CHANGE-LOG TEXT.	18 19 20 21 ²² ticket340.
33.	Sections 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 188, 562, and 711, and MPI-2.2 Sections 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 164, 433, and 513. MPI-2.2 errata: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX in C and Fortran for the C++ types bool, Complex <float>, Complex<double>, and Complex<long double="">, corresponding to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX, which are removed in MPI-3.0. TICKET NOT YET PASSED. CHANGE-LOG TEXT AS ON TICKET.</long></double></float>	23 24 25 26 27 28 29 30 31 32 ticket281. 33 34 ticket340.
34.	Sections 5.9.2 on pages 188. MPI_C_COMPLEX is added to the "Complex" reduction group. TICKET NOT YET PASSED. CHANGE-LOG TEXT AS ON TICKET.	³⁵ ³⁶ ³⁷ ticket230-B.
35.	Section 2.3 on page 10, and Sections 16.2.1, 16.2.2, 16.2.7 on pages 642, 644, and 659. The new mpi_08 Fortran module is introduced.	³⁸ ticket247-S. ³⁹ ticket248-T. ⁴⁰ ticket231-C.
36.	Section 2.5.1 on page 12, Section 16.2.3 on page 646, Section 16.2.2 on page 644, and Section 16.2.7 on page 659. Handles to opaque objects are defined as named types within the mpi_08 Fortran module. The handle types are also available through the mpi Fortran module.	41 42 43 44 45 ticket234-F.
37.	Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 16.2.1, 16.2.10, 16.2.11, 16.2.12, 16.2.13 on pages 642, 670, 672, 673, 676, and Sections 16.2.3, 16.2.2, 16.2.7 on pages 646, 644, 659.	46 ticket235-G. 47 ticket236-H. 48

	1 2 3 4 5 6	Within the mpi_08 Fortran module, choice buffers are defined as assumed-type and assumed-rank according to Fortran 2008 TR 29113 [41], and the compile-time constant MPI_SUBARRAYS_SUPPORTED is set to .TRUE With this, Fortran subscript triplets can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TR 29113 feature, the constant is set to .FALSE
ticket239-K. ticket243-O.	7	Section 2.6.2 on page 18, Section 16.2.2 on page 644, and Section 16.2.7 on page 659. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.
ticket229.1.	10 39. 11 12 13 14 15	Section 3.2.5 on page 34, Section 16.2.3 on page 646, Section 16.2.2 on page 644, Section 16.2.7 on page 659, and Section 16.3.5 on page 698. Within the mpi_08 Fortran module, the status is defined as TYPE(MPI_Status). New conversion routines are added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new type MPI_F08_status, and the external variables MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE
ticket38. ticket274. ticket229.2.	18 /0	are added. Section 3.2.6 on page 36, and Section 3.8 on page 71.
ticket229.4. ticket229.2.	20 21	MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE. Section 3.6 on page 49.
ticket237-I. ticket229.2.	23 24	In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument of MPI_BUFFER_DETACH is wrongly defined and the argument is therefore unused.
	²⁵ 26 27	Section 4.1 on page 91, Section 4.1.6 on page 112, and Section 16.2.15 on page 677. The Fortran alignments of basic datatypes are implementation dependent. It is recommended that they are computed according to BIND(C) derived types. If an array
:	28 29 30	of structures (in $C/C++$) or derived types (in Fortran) should be communicated, it is recommended that the user creates a portable datatype handle and applies additionally MPI_TYPE_CREATE_RESIZED to this datatype handle.
ticket252-W.		Sections 4.1.10, 5.9.5, 5.9.7, 6.7.4, 6.8, 8.3.1, 8.3.2, 8.3.3, 15.1, 16.2.9 on pages 119,
	33 34	195, 201, 288, 294, 358, 360, 362, 619, and 662. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords
	35	TYPE and FUNCTION. The new dummy argument names must be used because the
	36 37	mpi and mpi_08 modules guarantee keyword-based actual argument lists. The argument name type was changed into oldtype in MPI_TYPE_DUP, and into datatype
:	38	in the Fortran USER_FUNCTION of MPI_OP_CREATE, and in MPI_TYPE_SET_ATTR,
	39	MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME,
	40 41	MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, in the callback prototype defi- nition MPI_Type_delete_attr_function, and the predefined callback function
	42	MPI_TYPE_NULL_DELETE_FN; function was changed into user_fn in
	43	$MPI_OP_CREATE, into \ \mathsf{comm_errhandler_fn \ in \ MPI_COMM_CREATE_ERRHANDLER,}$
	44	into win_errhandler_fn in MPI_WIN_CREATE_ERRHANDLER, into file_errhandler_fn in
ticket251-V.	45 46	MPI_FILE_CREATE_ERRHANDLER, into handler_fn in MPI_ERRHANDLER_CREATE. For consistency reasons, INOUBUF was changed into INOUTBUF in
	47	MPI_REDUCE_LOCAL, and intracomm into newintracomm in
	48	MPI_INTERCOMM_MERGE.

ticket 245-Q.	1	
	44. Section 8.2 on page 353. In Fortran with the mpi and mpi_f08 modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointer instead of only returning an address-sized integer that may be usable together a with non-standard Cray-pointer. The Fortran interfaces with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR in the mpi module and the mpif.h include file are deprecated since MPI-3.0. 8 ti	cket237-I.
	45. Section 16.2.15 on page 677, and Section 16.2.7 on page 659. Fortran SEQUENCE and BIND(C) derived application types can be used as buffers in MPI operations. ¹⁰ ti	icket238-J.
	46. Section 16.2.16 on page 679 to Section 16.2.19 on page 688, Section 16.2.7 on page 659, and Section 16.2.8 on page 661. The sections about Fortran optimization problems and their solution is partially rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the meaning of the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The Fortran routine MPI_F_SYNC_REG is added. To achieve a secure and portable programming interfaces, in Section 16.2.7, several requirements are defined for the combination of an MPI_library and a Fortran compiler to be MPI_3 0 compliant	cket229.1.
	47. Section 16.2.4 on page 649. 22 The use of the mpif h Fortran include file is strongly discouraged 23	icket232-D.
	48. Section 16.2.3 on page 646, and Section 16.2.7 on page 659. The existing mpi Fortran module must implement compile-time argument checking.	icket242-N.
	49. Section 16.2.2 on page 644. Within the mpi_08 Fortran module, dummy arguments are declared with INTENT=IN, OUT or INOUT as defined in the mpi_08 interfaces	icket230-B.
	50. Section 16.2.7 on page 659.31 tiThis new section summarizes requirements that an MPI library together with a Fortran32 ticompiler is compliant to the MPI standard33 ti	icket231-C. icket232-D. icket234-F. icket237-I.
	51. Section A.1.1, Table " <i>Predefined functions</i> " on page 720, Section A.1.3 on page 728, and Section A.3.4 on page 774. Within the new mpi_f08 module, all callback prototype definitions are defined with are light interfaces PROCEENUPE (MPI = -) with PLND(C) attribute	icket238-J. icket239-K. icket233-O. icket230-B. icket250-V.
	52. Section A.1.3 on page 728. 40 In some routines, the Fortran callback prototype names were changed fromFN to 41 FUNCTION to be consistent with the other language bindings. 42 43	
	B.2 TEST for Tickets 271, 168, 204, 280, 286, 287	
	This section is not part of the MPI standard and will be removed after the next meeting, July 2012. 46 47	cket-248T.

```
1
                MPI_Comm_dup_info(comm, info, newcomm, ierror) BIND(C)
           \mathbf{2}
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           3
                    TYPE(MPI_Info), INTENT(IN) :: info
           4
                    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
           5
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           6
                MPI_COMM_DUP_INFO(COMM, INFO, NEWCOMM, IERROR)
           7
                    INTEGER COMM, INFO, NEWCOMM, IERROR
           8
ticket-248T.
           9
                MPI_Comm_set_info(comm, info, ierror) BIND(C)
           10
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           11
                    TYPE(MPI_Info), INTENT(IN) :: info
           12
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           13
                MPI_COMM_SET_INFO(COMM, INFO, IERROR)
           14
                    INTEGER COMM, INFO, IERROR
ticket-248T. ^{15}
           16
                MPI_Comm_get_info(comm, info_used, ierror) BIND(C)
           17
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           18
                    TYPE(MPI_Info), INTENT(OUT) :: info_used
           19
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           20
                MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
           21
                    INTEGER COMM, INFO_USED, IERROR
          22
ticket-248T.
           23
                MPI_Win_set_info(win, info, ierror) BIND(C)
           24
                    TYPE(MPI_Win), INTENT(IN) :: win
           25
                    TYPE(MPI_Info), INTENT(IN) :: info
           26
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           27
                MPI_WIN_SET_INFO(WIN, INFO, IERROR)
           28
                    INTEGER WIN, INFO, IERROR
           29
ticket-248T.
           30
                MPI_Win_get_info(win, info_used, ierror) BIND(C)
           ^{31}
                    TYPE(MPI_Win), INTENT(IN) :: win
           32
                    TYPE(MPI_Info), INTENT(OUT) :: info_used
           33
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           34
                MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
           35
                    INTEGER WIN, INFO_USED, IERROR
           36
ticket-248T.
           37
                MPI_Comm_idup(comm, newcomm, request, ierror) BIND(C)
           38
                    TYPE(MPI_Comm), INTENT(IN) :: comm
           39
                    TYPE(MPI_Comm), ASYNCHRONOUS :: newcomm
           40
                    TYPE(MPI_Request), INTENT(OUT) :: request
           41
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           42
                MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
           43
                    INTEGER COMM, NEWCOMM, REQUEST, IERROR
           44
ticket-248T. _{45}
               MPI_Get_library_version(version, resulten, ierror) BIND(C)
           46
                    CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
           47
                    INTEGER, INTENT(OUT) :: resultlen
           48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR)	2 3
CHARACTER*(*) VERSION	4
INTEGER RESULTLEN, IERROR	5 ticket-248T.
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,	6
oldtype, newtype, ierror) BIND(C)	7
INTEGER, INTENT(IN) :: count, blocklength	8
INTEGER(kind=MPI_Address_kind), INTENT(IN) ::	9 10
array_of_displacements(count) TYPE(MPI_Datatype), INTENT(IN) :: oldtype	11
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
	14
MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	15
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	16
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	17
	$^{18}_{19}$ ticket-248T.
<pre>MPI_Comm_create_group(comm, group, tag, newcomm, ierror) BIND(C) TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	20
TYPE(MPI_Group), INTENT(IN) :: group	21
INTEGER, INTENT(IN) :: tag	22
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)	25 26
INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR	07
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) BIND(C)	$^{27}_{28}$ ticket-248T.
TYPE(MPI_Comm), INTENT(IN) :: comm	29
INTEGER, INTENT(IN) :: split_type, key	30
TYPE(MPI_Info), INTENT(IN) :: info	31
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	32 33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	35
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR	36
	37
R.2. Channes from Mansien 2.1 to Mansien 2.2	38
B.3 Changes from Version 2.1 to Version 2.2	39
1. Section 2.5.4 on page 15.	40 41
It is now guaranteed that predefined named constant handles (as other constants)	42
can be used in initialization expressions or assignments, i.e., also before the call to	43
MPI_INIT.	44
2. Section 2.6 on page 17, Section 2.6.4 on page 20, and Section 16.1 on page 629.	45
The $C++$ language bindings have been deprecated and may be removed in a future	46
version of the MPI specification.	47
	48

2 3 4 5 6	3.	Section 3.2.2 on page 29. MPI_CHAR for printable characters is now defined for C type char (instead of signed char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed for predefined reduction operations.
7 8 9 10 11	4.	Section 3.2.2 on page 29. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.
12 13 14 15 16 17	5.	Section 3.4 on page 41, Section 3.7.2 on page 54, Section 3.9 on page 80, and Section 5.1 on page 151. The read access restriction on the send buffer for blocking, non blocking and collective API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.
18 19	6.	Section 3.7 on page 52. The Advice to users for IBSEND and IRSEND was slightly changed.
20 21 22 23	7.	Section 3.7.3 on page 58. The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.
24 25 26	8.	Section 3.7.6 on page 70. MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.
	9.	Section 5.8 on page 180. "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intracommunicators.
30 31 1 32 33 34	0.	Section 5.9.2 on page 188. Predefined parameterized datatypes (e.g., returned by MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g. MPI_REAL8) have been added to the list of valid datatypes in reduction operations.
35 1 36 37 38 39 40 41	1.	Section 5.9.2 on page 188. MPI_(U)INT{8,16,32,64}_T are all considered C integer types for the purposes of the predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran integer types. MPI_C_BOOL is considered a Logical type. MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.
	2.	Section 5.9.7 on page 201. The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.

 13. Section 5.10.1 on page 203.
 The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI standard.

14.	Section 5.11.2 on page 207. Added in place argument to MPI_EXSCAN.	1
	Added in place argument to MFT_EASCAN.	3
15.	Section $6.4.2$ on page 252, and Section 6.6 on page 268.	4
	Implementations that did not implement MPI_COMM_CREATE on intercommuni-	5
	cators will need to add that functionality. As the standard described the behav-	6
	ior of this operation on intercommunicators, it is believed that most implementa-	7
	tions already provide this functionality. Note also that the C++ binding for both	8
	MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.	9
		10
16.	Section $6.4.2$ on page 252 .	11
	MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm	12
	is an intracommunicator. If comm is an intercommunicator it was clarified that all	12
	processes in the same local group of comm must specify the same value for group.	14
1 🗖		14
17.	Section 7.5.4 on page 308.	
	New functions for a scalable distributed graph topology interface has been added.	16
	In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and	17
	MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++	18
	class Distgraphcomm were added.	19
18	Section $7.5.5$ on page 315 .	20
10.	For the scalable distributed graph topology interface, the functions	$^{21}_{22}$ ticket201.
	MPI_DIST_NEIGHBORS_COUNT]MPI_DIST_GRAPH_NEIGHBORS_COUNT and [$_{22}$ ticket201.
	MPI_DIST_NEIGHBORS]MPI_DIST_GRAPH_NEIGHBORS and the constant	
	MPI_DIST_GRAPH were added.	24
		25
19.	Section $7.5.5$ on page 315 .	26
	Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS	27
	and MPI_GRAPH_NEIGHBORS_COUNT.	28
		29
20.	Section 8.1.1 on page 349.	30
	The subversion number changed from 1 to 2.	31
91	Section 8.3 on page 356, Section 15.2 on page 626, and Annex A.1.3 on page 728.	32
41.	Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to	33
	MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.	34
		35
22.	Section 8.7.1 on page 378.	36
	Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Imple-	37
	mentors must now also register all implementation-internal attribute deletion callbacks	38
	on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.	39
		40
23.	Section 11.3.4 on page 441.	41
	The restriction added in MPI 2.1 that the operation MPI_REPLACE in	42 43
	MPI_ACCUMULATE can be used only with predefined datatypes has been removed.	
	MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0.	44
	Also, a clarification has been made that MPI_REPLACE can be used only in	45
	MPI_ACCUMULATE, not in collective operations that do reductions, such as	
	MPI_REDUCE and others.	47
		48

Unofficial Draft for Comment Only

1	24.	Section 12.2 on page 493 .
2		Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for
3		MPI::Grequest::Start() for consistency with the rest of MPI functions that take function
4		pointer arguments.
5	05	
6	25.	Section 13.5.2 on page 560, and Table 13.2 on page 562.
7		MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX,
8		MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX,
9		MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes
10		in the external32 representation.
11	26.	Section $16.3.7$ on page 704.
12		The description was modified that it only describes how an MPI implementation be-
13		haves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example
14		
15		16.17 was replaced with three new examples 16.25, 16.26, and 16.27 on pages 705-707
16		explicitly detailing cross-language attribute behavior. Implementations that matched
		the behavior of the old example will need to be updated.
17		
18	27.	Annex A.1.1 on page 711.
19		Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 727).
20		
21	28.	Annex A.1.1 on page 711. Table Named Predefined Datatypes.
22		Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,
23		MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and
		MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.
24		win i ce con de lo com e com e ca a ca a a predemieu datatypes.
25		
26	B.4	Changes from Version 2.0 to Version 2.1
27		
28	1.	Section 3.2.2 on page 29, Section 16.1.6 on page 633, and Annex A.1 on page 711.
29		In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn-
30		
		onym for MPI_LONG_LONG_INT.
31	2	Section 3.2.2 on page 29, Section 16.1.6 on page 633, and Annex A.1 on page 711.
32	۷.	
33		\mathbf{M} \mathbf{D} \mathbf{L} (\mathbf{M}) (\mathbf{L}) \mathbf{M} (\mathbf{M}) \mathbf{M} \mathbf{M} \mathbf{D} \mathbf{L} (\mathbf{M}) (\mathbf{L}) (\mathbf{M})
9.4		MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym),
34		MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved
34 35		
	2	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings.
35 36	3.	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34.
35 36 37	3.	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings.
35 36 37 38	3.	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34.
35 36 37	3.	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes
35 36 37 38	3.	 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero,
35 36 37 38 39	3.	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes
35 36 37 38 39 40		MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.
35 36 37 38 39 40 41		MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned. Section 4.1 on page 91.
35 36 37 38 39 40 41 42 43		 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned. Section 4.1 on page 91. General rule about derived datatypes: Most datatype constructors have replication
35 36 37 38 39 40 41 42 43 44		 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned. Section 4.1 on page 91. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the
35 36 37 38 39 40 41 42 43 44 45		 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned. Section 4.1 on page 91. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on
35 36 37 38 39 40 41 42 43 44		 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned. Section 4.1 on page 91. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the
35 36 37 38 39 40 41 42 43 44 45		 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings. Section 3.2.5 on page 34. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned. Section 4.1 on page 91. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on

5.	Section 4.3 on page 147.	1
	MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.	2 3
0		4
6.	Section 5.9.6 on page 199. If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should pro-	5
	vide count and datatype arguments that specify the same type signature (i.e., it is not	6
	necessary that both groups provide the same count value).	7 8
7.	Section $6.3.1$ on page 242 .	9
	MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL	10 11 12
	as the translated rank.	12
8	Section 6.7 on page 276 .	14
0.	About the attribute caching functions:	15
	Advice to implementors. High-quality implementations should raise an er-	16 17
	ror when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL	18
	is used with an object of the wrong type with a call to	19
	MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or	20
	MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each key-	21
	val, information on the type of the associated user function. (End of advice to	22
	implementors.)	23
9.	Section 6.8 on page 294.	24
	In MPI_COMM_GET_NAME: In C, a null character is additionally stored at	25
	name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For-	26
	tran, name is padded on the right with blank characters. resultlen cannot be larger	27 28
	then MPI_MAX_OBJECT_NAME.	20
10	Section 7.4 on page 302 .	30
10.	About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must	31
	have identical values on all processes of the group of comm_old.	32
		33
11.	Section 7.5.1 on page 304.	34
	In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology	35
	is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.	36
	of it fulling is negative.	37
12.	Section $7.5.3$ on page 306 .	38 39
	In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$, then	40
	MPI_COMM_NULL is returned in all processes.	41
13.	Section $7.5.3$ on page 306 .	42
	In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times	43
	in the list of neighbors of a process (i.e., there may be multiple edges between two	44
	processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the	45
	graph). The adjacency matrix is allowed to be non-symmetric.	46
		47
		48

1 2 3		Advice to users. Performance implications of using multiple edges or a non- symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (<i>End of advice to users.</i>)
4 5 6 7 8]	Section 7.5.5 on page 315. In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero- dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.
9 10 11 12]	Section 7.5.5 on page 315. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol- ogy, coord is not significant and 0 is returned in rank.
13 14 15]	Section 7.5.5 on page 315. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.
16 17 18 19 20 21] i	Section 7.5.6 on page 323. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.
22 23 24 25 26] 8	Section 7.5.7 on page 325. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associ- ated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.
27 28		Section 8.1.1 on page 349. The subversion number changed from 0 to 1.
29 30 31 32 33 34]]	Section 8.1.2 on page 351. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.
35 36 37 38 39 40 41	 	Section 8.3 on page 356. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.
42 43 44 45 46 47 48	 	Section 8.7 on page 373, see explanations to MPI_FINALIZE. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is col- lective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 416.

22. Section 8.7 on page 373. About MPI_ABORT:

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

23. Section 9 on page 383.

An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

- 24. Section 11.3 on page 435. MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE, MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI pointto-point communication. See also item 25 in this list.
- 25. Section 11.3 on page 435. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.
- 26. Section 11.3.4 on page 441. MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
- 27. Section 13.2.8 on page 521. About MPI_FILE_SET_VIEW and MPI_FILE_SET_INFO: When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.
- 28. Section 13.2.8 on page 521. About MPI_FILE_GET_INFO: If no hint exists for the file associated with fh, a handle to a newly created info object is returned that contains no key/value pair.
 20. Section 12.2 on page 524.
- 29. Section 13.3 on page 524. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
 30. Section 13.5.2 on page 560.
 - The bias of 16 byte doubles was defined with 10383. The correct value is 16383.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

 31

 $45 \\ 46$

1 2 2	31.	Section 16.1.4 on page 630. In the example in this section, the buffer should be declared as const void* buf.
3 4 5	32.	Section 16.2.9 on page 662. About MPI_TYPE_CREATE_F90_xxxx:
6 7 8 9 10 11 12 13 14 15 16		Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_xxxx with the same combination of $(xxxx,p,r)$. The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_xxxx and using a hash-table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (xxxx,p,r). (End of advice to implementors.)
17 18 19	33.	Section A.1.1 on page 711. MPI_BOTTOM is defined as void * const MPI::BOTTOM.
20 21		
22 23		
24 25		
26		
27		
28		
29 30		
31		
32		
33		
34		
35		
36		
37 38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

Bibliography

- V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1
- [4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.
 1.2
- [5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990. 1.2
- [6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. IJHPCN, 1(1/2/3):91–99, 2004. 11.7
- [7] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. 13.1
- [8] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992. 1.2
- [9] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2
- [10] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. 1.2
- [11] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormholerouted networks for interprocessor communication in multicomputers. In *Proceedings* of the 1990 Supercomputing Conference, pages 647–656, 1990. 7.1

9 10

11

12

13 14

15

16

17 18

19

20

21

22

23 24

25

26

27 28

29

30

 31

32

33 34

35

36 37

38

39

40

41

42

43

 $\frac{44}{45}$

46

47

2 3	[12]	S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In <i>Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)</i> , pages II–1 – II–4, 1991. 7.1
4 5 6	[13]	Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 1.2, 7.4
7 8 9 10 11	[14]	Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In <i>IPPS '93 Workshop on Input/Output in Parallel Computer Systems</i> , pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38. 13.1
12 13 14	[15]	J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. <i>Computers in Physics</i> , 7(2):166–75, April 1993. 1.2
15 16 17 18	[16]	J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user- level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993. 1.2
19 20 21	[17]	Edinburgh Parallel Computing Centre, University of Edinburgh. <i>CHIMP Concepts</i> , June 1991. 1.2
22 23	[18]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
24 25 26	[19]	Margaret A. Ellis and Bjarne Stroustrup. The Annotated $C++$ Reference Manual. Addison Wesley, 1990.
27 28 29 30	[20]	D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 6.1.2
31 32 33	[21]	C++ Forum. Working paper for draft proposed international standard for information systems — programming language C++. Technical report, American National Standards Institute, 1995.
34 35 36 37	[22]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The In- ternational Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 1.3
38 39 40	[23]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3
41 42 43	[24]	Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing. MIT Press, 1994. 10.1
44 45 46 47 48	[25]	G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable in- strumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2

- [26] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing Probe for Multi-Threaded MPI Applications. Technical Report 674, Indiana University, Jan. 2009. 3.8.2
- [27] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2
- [28] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90. 16.2.3, 16.2.3
- [29] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI Support for Advanced Hybrid Programming Models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 3.8, 3.8.2
- [30] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12
- [31] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse Non-Blocking Collectives in Quantum Mechanical Calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6
- [32] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing To Thread or not to Thread? In Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society, Oct. 2008. 5.12
- [33] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07.* IEEE Computer Society/ACM, Nov. 2007. 5.12
- [34] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication Optimization for Medical Image Reconstruction Algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12
- [35] T. Hoefler and J. L. Traeff. Sparse Collective Operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 7.6
- [36] Institute of Electrical and Electronics Engineers, New York. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, 1985. 13.5.2
- [37] International Organization for Standardization, Geneva, ISO 8859-1:1987. Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1, 1987. 13.5.2

Unofficial Draft for Comment Only

1

2

3

4

5 6

7

8

9

10

11

12

13

14 15

16

17

18

19 20

21

22

23

24 25

26

27 28

29

30

 31

32 33

34

35

36

37

38

39

40 41

42

43 44

45

46

1 2 3	[38]	International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language], December 1996. 12.4, 13.2.1
4 5 6 7	[39]	International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. In- formation technology – Programming languages – Fortran – Part 1: Base language, November 2010. 16.2.1, 16.2.2
8 9 10 11 12	[40]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113, Draft N1869. TR on further interoperability with C, July, 18 2011. http://www.nag.co.uk/sc22wg5/ and ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1869.pdf. 16.2.2
13 14 15	[41]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113. TR on further interoperability with C, 2012. http://www.nag.co.uk/sc22wg5/. 16.2.1, 16.2.1, 16.2.2, 16.2.7, 37
16 17 18	[42]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 4.1.4
19 20 21 22	[43]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994</i> Symposium on Operating Systems Design and Implementation, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1
23 24	[44]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 7.1
25 26 27 28 29 30	[45]	S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at http://www.netbsd.org/Documentation/lite2/psd/. 10.5.5
31 32	[46]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2
33 34	[47]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1
35 36 37	[48]	William J. Nitzberg. <i>Collective Parallel I/O</i> . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1
38 39 40	[49]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 10.5.5
41 42 43	[50]	Paul Pierce. The NX/2 operating system. In <i>Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications</i> , pages 384–390. ACM Press, 1988. 1.2
44 45 46 47 48	[51]	Martin Schulz and Bronis R. de Supinski. P^N MPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In ACM/IEEE Supercomputing Conference (SC), pages 1–10. ACM, 2007. 14.2.8

[52]	K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In <i>Proceedings of Supercomputing '95</i> , December 1995. 13.1	1 2
[53]	A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, <i>Proceedings of the Fifth Distributed Memory Concurrent Computing Conference</i> , pages 767–776. IEEE Press, 1990. 1.2, 6.1.2	3 4 5 6 7
[54]	A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2	8 9 10 11
[55]	Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, <i>Proceedings of the Scalable</i> <i>Parallel Libraries Conference</i> , pages 166–173. IEEE Computer Society Press, October 1993. 6.1	12 13 14 15
[56]	Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU- 940722, Mississippi State University — Dept. of Computer Science, April 1994. http://www.erc.msstate.edu/mpi/mpix.html. 5.2.2	16 17 18 19 20
[57]	Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan E. Doss. Explicit parallel programming in C++ based on the message-passing interface (MPI). In Gregory V. Wilson, editor, <i>Parallel Programming Using C++</i> , Engineering Computation Series. MIT Press, July 1996. ISBN 0-262-73118-5.	21 22 23 24 25
[58]	Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. <i>Parallel Computing</i> , 20(4):565–596, April 1994. 6.1.2, 6.5.6	26 27 28
[59]	Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. <i>Scientific Programming</i> , 5(4):301–317, Winter 1996. 13.1	29 30 31 32
[60]	The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2	33 34
[61]	D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2	35 36 37 38
		39
		40
		41 42
		43
		44
		45
		46
		47
		48
	Unofficial Draft for Comment Only	

Examples Index

10

 11

12

 $13 \\ 14$

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C/C++ examples.

ASYNCHRONOUS, 690, 692 untyped, 3915Attributes between languages, 705 Deadlock 16if not buffered, 4817Basic tool using performance variables in with MPI_Bcast, 228, 229 18 the MPI tool information interface, wrong message exchange, 4819 612 20False matching of collective operations, 232 x [[illegalticket1820mmunicator handlet182. ticket1821 x [illegaltioketik82IND, 665 initialization, 637 Fortran 90 copying and sequence problem, 22 C++ declarations in mpi.h, 640 672, 674, 675 23 C++ deriving from C++ MPI class, 631 24Fortran 90 derived types, 677 C++ handle assignment and comparison 25Fortran 90 heterogeneous communication, operators, 63726669.670 C++ handle assignment operator, 632 Fortran 90 MPI_TYPE_MATCH_SIZE 27C++ handle scope destruction, 632 28 implementation, 668 C++ MPI class comparison operator, 631 29 Fortran 90 overlapping communication and C++ profiling example, 641, 642 computation, 691, 693 30 C/C++ handle conversion, 697, 698 31 Fortran 90 register optimization, 680–682 C/Fortran handle conversion, 696 32 Cartesian virtual topologies, 343 Independence of nonblocking operations, 33 Client-server code, 69, 70 23534 with blocking probe, 73 Intercommunicator, 255, 258 35 with blocking probe, wrong, 73 Interlanguage communication, 709 36 Intertwined matching pairs, 46 37 Datatype 38 3D array, 131 Message exchange, 4839 absolute addresses, 137 Mixing blocking and nonblocking collective 40 array of structures, 134 operations, 231 41 elaborate example, 145, 146 Mixing collective and point-to-point 42matching type, 120requests, 234 43 matrix transpose, 132, 133 MPI::Comm::Probe, 37 44 union, 138MPI_ACCUMULATE, 443 45Datatypes MPI_Accumulate, 487, 489 46 matching, 38 MPI_ADDRESS, 111 47not matching, 38 MPI_Address, 134, 137, 138, 145 48

MPI_Aint, 134 MPI_Allgather, 179 MPI_ALLOC_MEM, 355 MPI_Alloc_mem, 356, 489 MPI_ALLREDUCE, 200 MPI_Alltoall, 233 MPI_Barrier, 375, 476–479, 485, 487, 488 MPI_Bcast, 159, 213, 228-232 MPI_BSEND, 46 MPI_Buffer_attach, 50, 375 MPI_Buffer_detach, 50 MPI_BYTE, 39 MPI_Cancel, 375 MPI_CART_COORDS, 324 MPI_CART_GET, 343 MPI_CART_RANK, 324, 343 MPI_CART_SHIFT, 324, 343 MPI_CART_SUB, 325 MPI_CHARACTER, 39 MPI_Comm_create, 255 MPI_Comm_group, 255 MPI_Comm_remote_size, 258 MPI_COMM_SPAWN, 394 MPI_Comm_spawn, 394 MPI_COMM_SPAWN_MULTIPLE, 400 MPI_Comm_spawn_multiple, 400 MPI_Comm_split, 258 MPI_Compare_and_swap, 488, 489 MPI_DIMS_CREATE, 306, 343 MPI_DIST_GRAPH_CREATE, 313 MPI_Dist_graph_create, 314 MPI_DIST_GRAPH_CREATE_ADJACENT 313MPI_FILE_CLOSE, 535, 539 MPI_FILE_GET_AMODE, 520 MPI_FILE_IREAD, 539 MPI_FILE_OPEN, 535, 539 MPI_FILE_READ, 535 MPI_FILE_SET_ATOMICITY, 572 MPI_FILE_SET_VIEW, 535, 539 MPI_FILE_SYNC, 573 MPI_Finalize, 375, 376 MPI_FREE_MEM, 355 MPI_Free_mem, 489 MPI_Gather, 146, 163, 164, 169 MPI_Gatherv, 146, 165–169 MPI_GET, 439, 441 MPI_Get, 476–478, 484, 485

	MPI_Get_accumulate, 487, 489	1
	MPI_GET_ADDRESS, 111, 677, 678, 701	2
	MPI_Get_address, 134, 137, 138, 145	3
	MPI_GET_COUNT, 122	4
	MPI_GET_ELEMENTS, 122	5
	MPI_GRAPH_CREATE, 307, 320	6
	MPI_GRAPH_NEIGHBORS, 320	7
	MPI_GRAPH_NEIGHBORS_COUNT,	8
	320	9
	MPI_Grequest_complete, 498	10
	MPI_Grequest_start, 498	11
	MPI_Group_free, 255	12
	MPI_Group_incl, 255	13
	MPI_Iallreduce, 234	14
	MPI_Ialltoall, 233	15
	MPI_Ibarrier, 231–234	16
	MPI_Ibcast, 234, 235	17
	MPI_Iprobe, 375	18
	MPI_IRECV, 60–62, 69, 70	19
	MPI_Irecv, 234	20
	MPI_ISEND, 60–62, 69, 70	21
	MPI_Op_create, 198, 199, 208	22
	MPI_Pack, 144–146	23
	MPI_Pack_size, 146	24
	MPI_PROBE, 73	25
	MPI_Put, 460, 466, 477–479, 484, 485	26
	MPI_RECV, 38, 39, 46, 48, 62, 73, 120	27
	MPI_Recv, 233	28
	MPI_REDUCE, 189, 190, 193	29
	MPI_Reduce, 192, 194, 198, 199	30
	MPI_REQUEST_FREE, 61	31
Γ,	MPI_Request_free, 375	32
,	MPI_Rget, 488	33
	MPI_Rput, 488	34
	MPI_Scan, 208	35
	MPI_Scatter, 173	36
	MPI_Scatterv, 174	37
	MPI_SEND, 38, 39, 48, 62, 73, 120	38
	MPI_Send, 134, 137, 138, 145, 233, 234	39
	MPI_SENDRECV, 131–133	40
	MPI_SENDRECV_REPLACE, 324	41
	MPI_SSEND, 46, 62	42
	MPI_Test_cancelled, 375	43
	MPI_TYPE_COMMIT, 118, 131–133, 439,	44
	677, 678	45
	MPI_Type_commit, 134, 137, 138, 145,	46
	164-169, 174, 208	47
		48

1	MPI_TYPE_CONTIGUOUS, 94, 113, 120,	N
2	122	N
3	MPI_Type_contiguous, 164	Ν
4	MPI_TYPE_CREATE_DARRAY, 110	N
5	MPI_TYPE_CREATE_HVECTOR, 131,	Ν
6	132	Ν
7	MPI_Type_create_hvector, 134, 137	Ν
8	MPI_TYPE_CREATE_INDEXED_BLOCK,	Ν
9	439	Ν
10	MPI_TYPE_CREATE_RESIZED, 677,	Ν
11	678	n
12	MPI_TYPE_CREATE_STRUCT, 102,	
13	113, 133, 677, 678	ľ
14	MPI_Type_create_struct, 134, 137, 138,	
15	145, 167, 169, 208	ľ
16	MPI_TYPE_CREATE_SUBARRAY, 580	
17	MPI_TYPE_EXTENT, 131–133, 439	N
18	MPI_Type_extent, 134	
19	MPI_TYPE_FREE, 439	N
20	MPI_Type_get_contents, 139	N
21	MPI_Type_get_envelope, 139	
22	MPI_TYPE_GET_EXTENT, 441, 443	
23	MPI_TYPE_HVECTOR, 131, 132	
24	MPI_Type_hvector, 134, 137	(
25	MPI_TYPE_INDEXED, 98, 132	F
26	MPI_Type_indexed, 134, 137	T
27	MPI_TYPE_STRUCT, 102, 113, 133	F
28	MPI_Type_struct, 134, 137, 138, 145, 167,	F
29	169, 208	T
30	MPI_TYPE_VECTOR, 95, 131–133	
31	MPI_Type_vector, 165, 166, 168, 174	F
32	MPI_Unpack, 145, 146	
33	MPI_User_function, 199	
34	MPI_WAIT, 60–62, 69, 70, 539	
35	MPI_Wait, 231–234	ſ
36	MPI_Waitall, 234, 488	ſ
37	MPI_WAITANY, 69	ן
38	MPI_Waitany, 488	т
39	MPI_WAITSOME, 70	Ţ
40	MPI_Win_attach, 489	
41	MPI_Win_complete, 460, 478, 479, 485	Ţ
42	MPI_WIN_CREATE, 439, 441, 443	
43	MPI_Win_create_dynamic, 489	
44	MPI_Win_detach, 489	
45	MPI_WIN_FENCE, 439, 441, 443	
46	MPI_Win_fence, 484	
47	MPI_Win_flush, 478, 487–489	
48	MPI_Win_flush_all, 487	
	,	

MPI_Win_flush_local, 477 MPI_WIN_FREE, 441, 443 MPI_Win_lock, 466, 476–479 MPI_Win_lock_all, 488, 489 MPI_Win_post, 478, 479, 485 MPI_Win_start, 460, 478, 479, 485 MPI_Win_sync, 477, 478, 487, 488 MPI_Win_unlock, 466, 476-479 MPI_Win_unlock_all, 488, 489 MPI_Win_wait, 478, 479, 485 mpiexec, 381 Neighborhood collective communication, 343 No Matching of Blocking and Nonblocking collective operations, 233 Non-deterministic program with MPI_Bcast, 230 Non-overtaking messages, 46 Nonblocking operations, 60, 61 message ordering, 62progress, 62

Overlapping Communicators, 234

- Pipelining nonblocking collective operations, 234 Profiling interface, 587
- Progression of nonblocking collective operations, 233
- Reading the value of a control variable in the MPI tool information interface, 601

Threads and MPI, 503 Topologies, 343 Typemap, 93–95, 98, 102, 110

Using MPI_T_CVAR_GET_INFO to list all names of control variables., 598

Virtual topologies, 343

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI::*_NULL, <u>631</u>	MPI::COMBINER_STRUCT_INTEGER,	13 14
MPI::_LONG_LONG, 633, 859	723	14
MPI::ANY_SOURCE, 713	MPI::COMBINER_SUBARRAY, 723	15
MPI::ANY_TAG, 713	MPI::COMBINER_VECTOR, 723	10
MPI::APPNUM, 722	MPI::COMM_NULL, 633, 637, 719	
MPI::ARGV_NULL, 725	MPI::COMM_SELF, 717	18
MPI::ARGVS_NULL, 725	MPI::COMM_WORLD, 717	19
MPI::BAND, 718	MPI::COMPLEX, 634, 636, 716, 863	20
MPI::BOOL, 634, 636, 716, 863	MPI::CONGRUENT, 718	21
MPI::BOR, 718	MPI::DATATYPE_NULL, 719	22
MPI::BOTTOM, 713	MPI::DISPLACEMENT_CURRENT, 724	23
MPI::BSEND_OVERHEAD, 713	MPI::DIST_GRAPH, 720	24
MPI::BXOR, 718	MPI::DISTRIBUTE_BLOCK, 724	25
MPI::BYTE, 633, 634, 636, 715, 716	MPI::DISTRIBUTE_CYCLIC, 724	26
MPI::CART, 720	MPI::DISTRIBUTE_DFLT_DARG, 724	27
MPI::CHAR, 634, 715	MPI::DISTRIBUTE_NONE, 724	28
MPI::CHARACTER, 634, 716	MPI::DOUBLE, 634, 636, 715	29
MPI::COMBINER_CONTIGUOUS, 723	MPI::DOUBLE_COMPLEX, 634, 636,	30
MPI::COMBINER_DARRAY, 723	716, 863	31
MPI::COMBINER_DUP, 723	MPI::DOUBLE_INT, 635, 717	32
MPI::COMBINER_F90_COMPLEX, 723	MPI::DOUBLE_PRECISION, 634, 636,	33
MPI::COMBINER_F90_INTEGER, 723	716	34
MPI::COMBINER_F90_REAL, 723	MPI::DUP_FN, 722	35
MPI::COMBINER_HINDEXED, 723	MPI::ERR_ACCESS, 712	36
MPI::COMBINER_HINDEXED_INTEGER,	MPI::ERR_AMODE, 712	37
723	MPI::ERR_ARG, 711	38
MPI::COMBINER_HVECTOR, 723	MPI::ERR_ASSERT, 712	39
MPI::COMBINER_HVECTOR_INTEGER,	MPI::ERR_BAD_FILE, 712	40
723	MPI::ERR_BASE, 712	41
MPI::COMBINER_INDEXED, 723	MPI::ERR_BUFFER, 711	42
MPI::COMBINER_INDEXED_BLOCK,	MPI::ERR_COMM, 711	43
723	MPI::ERR_CONVERSION, 712	44
MPI::COMBINER_NAMED, 723	MPI::ERR_COUNT, 711	45
MPI::COMBINER_RESIZED, 723	MPI::ERR_DIMS, 711	46
MPI::COMBINER_STRUCT, 723	MPI::ERR_DISP, 712	47
	,	48

1	MPI::ERR_DUP_DATAREP, 712
2	MPI::ERR_FILE, 712
3	MPI::ERR_FILE_EXISTS, 712
4	MPI::ERR_FILE_IN_USE, 712
5	MPI::ERR_GROUP, 711
6	MPI::ERR_IN_STATUS, 712
7	MPI::ERR_INFO, 712
8	MPI::ERR_INFO_KEY, 712
9	MPI::ERR_INFO_NOKEY, 712
10	MPI::ERR_INFO_VALUE, 712
11	MPI::ERR_INTERN, 711
12	MPI::ERR_IO, 712
13	MPI::ERR_KEYVAL, 712
14	MPI::ERR_LASTCODE, 712
15	MPI::ERR_LOCKTYPE, 712
16	MPI::ERR_NAME, 712
17	MPI::ERR_NO_MEM, 712
18	MPI::ERR_NO_SPACE, 712
19	MPI::ERR_NO_SUCH_FILE, 712
20	MPI::ERR_NOT_SAME, 712
21	MPI::ERR_OP, 711
22	MPI::ERR_OTHER, 711
23	MPI::ERR_PENDING, 711
24	MPI::ERR_PORT, 712
25	MPI::ERR_QUOTA, 712
26	MPI::ERR_RANK, 711
27	MPI::ERR_READ_ONLY, 712
28	MPI::ERR_REQUEST, 711
29	MPI::ERR_RMA_CONFLICT, 712
30	MPI::ERR_RMA_SYNC, 712
31	MPI::ERR_ROOT, 711
32	MPI::ERR_SERVICE, 712
33	MPI::ERR_SIZE, 712
34	MPI::ERR_SPAWN, 712
35	MPI::ERR_TAG, 711
36	MPI::ERR_TOPOLOGY, 711
37	MPI::ERR_TRUNCATE, 711
38	MPI::ERR_TYPE, 711
39	MPI::ERR_UNKNOWN, 711
40	MPI::ERR_UNSUPPORTED_DATAREP,
41	712
42	MPI::ERR_UNSUPPORTED_OPERATIO
43	712
44	MPI::ERR_WIN, 712
45	MPI::ERRHANDLER_NULL, 719
46	MPI::ERRORS_ARE_FATAL, 20, 714
47	MPI::ERRORS_RETURN, 20, 21, 714
48	

MPI::ERRORS_THROW_EXCEPTIONS, 20, 21, 24, 357, 714 MPI::F_COMPLEX, 634, 636, 716 MPI::F_COMPLEX16, 635, 636, 859 MPI::F_COMPLEX32, 635, 636, 859 MPI::F_COMPLEX4, 635, 636, 859 MPI::F_COMPLEX8, 635, 636, 859 MPI::F_DOUBLE_COMPLEX, 635, 636, 716 MPI::FILE_NULL, 719 MPI::FLOAT, 634, 636, 715 MPI::FLOAT_INT, 635, 717 MPI::GRAPH, 720 MPI::GROUP_EMPTY, 719 MPI::GROUP_NULL, 719 **MPI::HOST**, **718** MPI::IDENT, 718 MPI::IN_PLACE, 713 MPI::INFO_NULL, 719 MPI::INT, 633, 634, 715 MPI::INTEGER, 633, 634, 716 MPI::INTEGER1, 635, 636, 716 MPI::INTEGER16, 635, 636, 859 MPI::INTEGER2, 635, 636, 716 MPI::INTEGER4, 635, 636, 716 MPI::INTEGER8, 635, 636, 716 MPI::IO, 718 MPI::KEYVAL_INVALID, 713 MPI::LAND, 718 MPI::LASTUSEDCODE, 722 MPI::LB, 717 MPI::LOCK_EXCLUSIVE, 713 MPI::LOCK_SHARED, 713 MPI::LOGICAL, 634, 636, 716 MPI::LONG, 633, 634, 715 MPI::LONG_DOUBLE, 634, 636, 715 MPI::LONG_DOUBLE_COMPLEX, 634, 636, 716, 863 MPI::LONG_DOUBLE_INT, 635, 717 MPI::LONG_INT, 635, 717 MPI::LONG_LONG, 634, 715, 859 ION, MPI::LONG_LONG_INT, 715 MPI::LOR, 718 MPI::LXOR, 718 MPI::MAX, 718 MPI::MAX_DATAREP_STRING, 714 MPI::MAX_ERROR_STRING, 714 MPI::MAX_INFO_KEY, 714

MPI::MAX_INFO_VAL, 714 MPI::MAX_OBJECT_NAME, 714 MPI::MAX_PORT_NAME, 714 MPI::MAX_PROCESSOR_NAME, 714 MPI::MAXLOC, 636, 718 MPI::MIN, 718 MPI::MINLOC, 636, 718 MPI::MODE_APPEND, 723 MPI::MODE_CREATE, 723 MPI::MODE_DELETE_ON_CLOSE, 723 MPI::MODE_EXCL, 723 MPI::MODE_NOCHECK, 723 MPI::MODE_NOPRECEDE, 723 MPI::MODE_NOPUT, 723 MPI::MODE_NOSTORE, 723 MPI::MODE_NOSUCCEED, 723 MPI::MODE_RDONLY, 723 MPI::MODE_RDWR, 723 MPI::MODE_SEQUENTIAL, 723 MPI::MODE_UNIQUE_OPEN, 723 MPI::MODE_WRONLY, 723 MPI::NULL_COPY_FN, 722 MPI::NULL_DELETE_FN, 722 MPI::OP_NULL, 719 MPI::ORDER_C, 724 MPI::ORDER_FORTRAN, 724 MPI::PACKED, 633, 634, 715, 716 MPI::PROC_NULL, 713 MPI::PROD, 718 MPI::REAL, 634, 636, 716 MPI::REAL16, 635, 636, 859 MPI::REAL2, 635, 636, 716 MPI::REAL4, 635, 636, 716 MPI::REAL8, 635, 636, 716 MPI::REPLACE, 718 MPI::REQUEST_NULL, 719 MPI::ROOT, 713 MPI::SEEK_CUR, 724 MPI::SEEK_END, 724 MPI::SEEK_SET, 724 MPI::SHORT, 633, 634, 715 MPI::SHORT_INT, 635, 717 MPI::SIGNED_CHAR, 633, 634, 715 MPI::SIMILAR, 718 MPI::SUCCESS, 711 MPI::SUM, 718 MPI::TAG_UB, 718 MPI::THREAD_FUNNELED, 724

MPI::THREAD_MULTIPLE, 724	1
MPI::THREAD_SERIALIZED, 724	2
MPI::THREAD_SINGLE, 724	3
MPI::TWODOUBLE_PRECISION, 635,	4
717	5
MPI::TWOINT, 635, 717	6
MPI::TWOINTEGER, 635, 717	7
MPI::TWOREAL, 635, 717	8
MPI::TYPECLASS_COMPLEX, 724	9
MPI::TYPECLASS_OOMI LEA, 724	10
MPI::TYPECLASS_REAL, 724	11
*	12
MPI::UB, 717	12
MPI::UNDEFINED, 713	
MPI::UNEQUAL, 718	14
MPI::UNIVERSE_SIZE, 722	15
MPI::UNSIGNED, 633, 634, 715	16
MPI::UNSIGNED_CHAR, 633, 634, 715	17
MPI::UNSIGNED_LONG, 633, 634, 715	18
MPI::UNSIGNED_LONG_LONG, 633,	19
634, 715	20
MPI::UNSIGNED_SHORT, 633, 634, 715	21
MPI::WCHAR, 634, 715	22
MPI::WIN_BASE, 722	23
MPI::WIN_DISP_UNIT, 722	24
MPI::WIN_NULL, 719	25
MPI::WIN_SIZE, 722	26
MPI::WTIME_IS_GLOBAL, 718	27
MPI_2DOUBLE_PRECISION, 192, 717	28
MPI_2INT, 192, 717	29
MPI_2INTEGER, 192, 717	30
MPI_2REAL, 192, 717	31
MPI_ADDRESS_KIND, 15, 16, <u>16</u> , 31,	32
126, 671, 704, 705, 714	33
	34
MPI_AINT, 31, 189, 715, 716, 868, 870	35
MPI_ANY_SOURCE, 32, 33, 46, 57, 58,	36
71, 72, 74–76, 83, 86, 88, 300, 352,	
	37
MPI_ANY_TAG, 15, 32, 33, 36, 57, 58, 71,	38
72, 74-77, 83, 86, 88, 89, 713	39
MPI_APPNUM, 415, 416, 722	40
MPI_ARGV_NULL, 16, 394, 395, 671, 725	41
MPI_ARGVS_NULL, 16, 399, 671, 725	42
MPI_ASYNC_PROTECTS_NONBLOCKING	43
16,643,644,647,650,651,660,	44
662,683,713,865	45
MPI_BAND, 188, 189, 718	46
MPI_BOR, 188, 189, 718	47
MPI_BOTTOM, 10, 16, 17, 37, 110, 123,	48

1	155 911 919 906 490 494 497
2	155, 311, 313, 396, 430, 434, 437,
3	645, 648, 656, 671, 676, 679, 681,
	682, 684-686, 688, 701, 702, 709,
4	713, 874
5	MPI_BSEND_OVERHEAD, 52, 353, 713
6	MPI_BXOR, 188, 189, 718
7	MPI_BYTE, 29, 30, 38–40, 147, 189, 512,
8	558, 571, 633, 709, 715, 716, 871
9	MPI_C_BOOL, 30, 189, 715, 859, 868, 870
10	MPI_C_COMPLEX, 30, 715, 863, 868, 870
11	MPI_C_DOUBLE_COMPLEX, 30, 189,
12	715,868,870
13	MPI_C_FLOAT128_COMPLEX, 863
14	MPI_C_FLOAT_COMPLEX, 30, 189, 715,
15	868, 870
16	MPI_C_LONG_DOUBLE_COMPLEX,
17	30, 31, 189, 715, 868, 870
18	MPI_C_QUAD_COMPLEX, 863
19	MPI_CART, 315, 720
20	MPI_CHAR, 30, 41, 103, 190, 191, 594,
21	715,868
22	MPI_CHARACTER, 29, 39–41, 190, 191,
23	716
24	MPI_COMBINER_CONTIGUOUS, 125,
25	129, 723
26	MPI_COMBINER_DARRAY, 125, 130,
27	723
28	MPI_COMBINER_DUP, 125, 129, 723
29	MPI_COMBINER_F90_COMPLEX, 125,
30	131, 723
31	MPI_COMBINER_F90_INTEGER, 125,
32	131, 723
33	MPI_COMBINER_F90_REAL, 125, 131,
34	723
35	MPI_COMBINER_HINDEXED, 125, 126,
36	130, 723
37	MPI_COMBINER_HINDEXED_BLOCK,
38	130, 723, 862
39	MPI_COMBINER_HINDEXED_INTEGER,
40	125, 130, 723, 862
41	MPI_COMBINER_HVECTOR, 125, 126,
42	129, 723
43	MPI_COMBINER_HVECTOR_INTEGER,
44	125, 126, 129, 723, 862
45	MPI_COMBINER_INDEXED, 125, 129,
46	723
47	MPI_COMBINER_INDEXED_BLOCK,
48	125, 130, 723
	, ,

MPI_COMBINER_NAMED, 125, 126, 129,723 MPI_COMBINER_RESIZED, 125, 131, 723 MPI_COMBINER_STRUCT, 125, 126, 130, 723 MPI_COMBINER_STRUCT_INTEGER, 125, 130, 723, 862 MPI_COMBINER_SUBARRAY, 125, 130, 723 MPI_COMBINER_VECTOR, 125, 129, 723 MPI_COMM_CREATE_KEYVAL, 280 MPI_COMM_NULL, 241, 254, 255, 257, 258, 260, 296, 305, 307, 397, 417, 418, 719, 871 MPI_COMM_PARENT, 296 MPI_COMM_SELF, 241, 277, 296, 378, 417, 514, 717, 869 MPI_COMM_TYPE_SHARED, 717, 862 MPI_COMM_WORLD, 15, 25, 32, 241-243, 250, 252, 263, 272, 273, 296, 305, 351, 352, 356, 359, 368, 376, 378, 380, 389, 390, 392, 393, 397, 399, 414-417, 506, 557, 576, 577, 600, 608, 695, 708, 717, 872 MPI_COMPLEX, 29, 189, 560, 663, 716 MPI_COMPLEX16, 189, 716 MPI_COMPLEX32, 189, 716 MPI_COMPLEX4, 189, 716 MPI_COMPLEX8, 189, 716 MPI_CONGRUENT, 252, 270, 718 MPI_CONVERSION_FN_NULL, 566 MPI_COUNT, 31, 189, 594, 715, 716, 861 MPI_COUNT_KIND, 16, 31, 714 MPI_CXX_BOOL, 863 MPI_CXX_DOUBLE_COMPLEX, 863 MPI_CXX_FLOAT_COMPLEX, 863 MPI_CXX_LONG_DOUBLE_COMPLEX, 863 MPI_DATATYPE, 21 MPI_DATATYPE_NULL, 118, 719 MPI_DISPLACEMENT_CURRENT, 525, 724, 873 MPI_DIST_GRAPH, 315, 720, 869 MPI_DISTRIBUTE_BLOCK, 107, 108, 724

MPI_DISTRIBUTE_CYCLIC, 107, 108,

724MPI_DISTRIBUTE_DFLT_DARG, 107, 108,724MPI_DISTRIBUTE_NONE, 107, 108, 724 MPI_DOUBLE, 30, 189, 594, 604, 662, 715 MPI_DOUBLE_COMPLEX, 29, 189, 560, 663, 716 MPI_DOUBLE_INT, 192, 717 MPI_DOUBLE_PRECISION, 29, 189, 663, 716 MPI_DUP_FN, 722 MPI_ERR_ACCESS, 367, 517, 578, 712 MPI_ERR_AMODE, 367, 515, 578, 712 MPI_ERR_ARG, 366, 711 MPI_ERR_ASSERT, 366, 471, 712 MPI_ERR_BAD_FILE, 367, 578, 712 MPI_ERR_BASE, 355, 366, 471, 712 MPI_ERR_BUFFER, 366, 711 MPI_ERR_COMM, 366, 711 MPI_ERR_CONVERSION, 367, 566, 578, 712MPI_ERR_COUNT, 366, 711 MPI_ERR_DIMS, 366, 711 MPI_ERR_DISP, 366, 471, 712 MPI_ERR_DUP_DATAREP, 367, 563, 578.712 MPI_ERR_FILE, 367, 578, 712 MPI_ERR_FILE_EXISTS, 367, 578, 712 MPI_ERR_FILE_IN_USE, 367, 517, 578, 712MPI_ERR_GROUP, 366, 711 MPI_ERR_IN_STATUS, 35, 37, 58, 66, 68, 358, 366, 497, 530, 639, 712 MPI_ERR_INFO, 366, 712 MPI_ERR_INFO_KEY, 366, 385, 712 MPI_ERR_INFO_NOKEY, 366, 385, 712 MPI_ERR_INFO_VALUE, 366, 385, 712 MPI_ERR_INTERN, 366, 711 MPI_ERR_IO, 367, 578, 712 MPI_ERR_KEYVAL, 292, 366, 712 MPI_ERR_LASTCODE, 365, 367, 368, 370, 712 MPI_ERR_LOCKTYPE, 366, 471, 712 MPI_ERR_NAME, 366, 411, 712 MPI_ERR_NO_MEM, 354, 366, 712 MPI_ERR_NO_SPACE, 367, 578, 712 MPI_ERR_NO_SUCH_FILE, 367, 517, 578, 712

MPI_ERR_NOATOMIC, 611	1
MPI_ERR_NOT_SAME, 367, 578, 712	2
MPI_ERR_OP, 366, 471, 711	3
MPI_ERR_OTHER, 365, 366, 711	4
MPI_ERR_PENDING, 66, 366, 711	5
MPI_ERR_PORT, 366, 408, 712	6
MPI_ERR_QUOTA, 367, 578, 712	7
MPI_ERR_RANK, 366, 471, 711	8
MPI_ERR_READ_ONLY, 367, 578, 712	9
MPI_ERR_REQUEST, 366, 711	10
MPI_ERR_RMA_ATTACH, 471, 712	11
MPI_ERR_RMA_CONFLICT, 366, 471,	12
712	13
MPI_ERR_RMA_RANGE, 471, 712	14
MPI_ERR_RMA_SHARED, 471	15
MPI_ERR_RMA_SYNC, 366, 471, 712	16
MPI_ERR_RMA_WRONG_FLAVOR,	17
429, 471	18
MPI_ERR_ROOT, <u>366</u> , <u>711</u>	19
MPI_ERR_SERVICE, 366, 410, 712	20
MPI_ERR_SIZE, 366, 471, 712	21
MPI_ERR_SPAWN, 366, 395, 396, 712	22
MPI_ERR_TAG, 366, 711	23
MPI_ERR_TOPOLOGY, 366, 711	24
MPI_ERR_TRUNCATE, 366, 711	25
MPI_ERR_TYPE, 366, 711	26
MPI_ERR_UNKNOWN, 365, 366, 711	27
MPI_ERR_UNSUPPORTED_DATAREP,	28
367, 578, 712	29
MPI_ERR_UNSUPPORTED_OPERATION,	30
367, 578, 712	31
MPI_ERR_WIN, 366, 471, 712	32
MPI_ERRCODES_IGNORE, 16, 396, 671,	33
725	34
MPI_ERRHANDLER_NULL, 364, 719	35
MPI_ERROR, 34, 58, 210, 448, 714	36
MPI_ERROR_STRING, 365	37
MPI_ERRORS_ARE_FATAL, 357, 370,	38
371, 471, 576, 714	39
MPI_ERRORS_RETURN, 357, 371, 577,	40
708, 714	41
MPI_F08_STATUS_IGNORE, 699, 725,	42
864	43
MPI_F08_STATUSES_IGNORE, 699, 725,	44
864	45
MPI_F_STATUS_IGNORE, 699, 725	46
MPI_F_STATUSES_IGNORE, 699, 725	47
MPI_FILE_NULL, 516, 577, 719	48
,, , , , , , , , , , , , , ,	

1	MPI_FLOAT, 30, 103, 187, 189, 559, 715
2	MPI_FLOAT128, 863
3	MPI_FLOAT_INT, 12, 192, 717
4	MPI_GRAPH, 315, 720
5	MPI_GROUP_EMPTY, 240, 246, 247,
6	254, 255, 719
7	MPI_GROUP_NULL, 240, 249, 719
8	MPI_HOST, 351, 718
9	MPI_IDENT, 244, 252, 718
10	MPI_IN_PLACE, 16, 154, 183, 648, 671,
11	713
12	MPI_INFO_GET_ENV, 863
13	MPI_INFO_KEY, 863
14	MPI_INFO_NULL, 313, 388, 396, 406,
15	515, 517, 526, 719
16	MPI_INT, 12, 30, 92, 188, 559, 560, 594,
17	595, 598, 603, 607, 662, 708, 710,
18	715
19	MPI_INT16_T, 30, 188, 715, 868, 870
20	MPI_INT32_T, 30, 188, 715, 868, 870
21	MPI_INT64_T, 30, 188, 715, 868, 870
22	
23	MPI_INT8_T, 30, 188, 715, 868, 870
24	MPI_INTEGER, 29, 38, 189, 662, 663,
25	710, 716
26	MPI_INTEGER1, 29, 189, 716
27	MPI_INTEGER16, 189, 716
28	MPI_INTEGER2, 29, 189, 560, 716
29	MPI_INTEGER4, 29, 189, 716
30	MPI_INTEGER8, 189, 667, 716
31	MPI_INTEGER_KIND, 16, 126, 704, 714
32	MPI_IO, 351, 352, 718
33	MPI_KEYVAL_INVALID, 281–283, 713
	MPI_LAND, 188, 189, 718
34	MPI_LASTUSEDCODE, 368, 369, 722
35	MPI_LB, 18, 19, 105, 109, 113, 115–117,
36	119, 120, 559, 717, 862
37	MPI_LOCK_EXCLUSIVE, 464, 713
38	MPI_LOCK_SHARED, 464, 713
39	MPI_LOGICAL, 29, 189, 716
40	MPI_LONG, 30, 188, 715
41	MPI_LONG_DOUBLE, 30, 189, 715
42	MPI_LONG_DOUBLE_INT, 192, 717
43	MPI_LONG_INT, 192, 717
44	MPI_LONG_LONG, 30, 188, 715, 870
45	MPI_LONG_LONG_INT, 30, 188, 715,
46	870
47	MPI_LOR, 188, 189, 718
48	MPI_LXOR, 188, 189, 718

MPI_MAX, 187–189, 208, 718 MPI_MAX_DATAREP_STRING, 15, 527, 563, 714 MPI_MAX_ERROR_STRING, 15, 365, 370, 714 MPI_MAX_INFO_KEY, 15, 366, 383, 386, 714 MPI_MAX_INFO_VAL, 15, 366, 383, 714 MPI_MAX_LIBRARY_VERSION_STRING, 15, 350, 351, 714, 860 MPI_MAX_OBJECT_NAME, 15, 295-298, 714, 860, 871 MPI_MAX_PORT_NAME, 15, 405, 714 MPI_MAX_PROCESSOR_NAME, 15, 353, 714, 872 MPI_MAXLOC, 188, 191, 192, 195, 718 MPI_MESSAGE_NO_PROC, 860 MPI_MESSAGE_NULL, 77, 78, 719, 860 MPI_MIN, 188, 189, 718 MPI_MINLOC, 188, 191, 192, 195, 718 MPI_MODE_APPEND, 514, 515, 723 MPI_MODE_CREATE, 514, 515, 523, 723 MPI_MODE_DELETE_ON_CLOSE, 514-516, 723 MPI_MODE_EXCL, 514, 515, 723 MPI_MODE_NOCHECK, 465, 469, 470, 723 MPI_MODE_NOPRECEDE, 469, 470, 723 MPI_MODE_NOPUT, 469, 470, 723 MPI_MODE_NOSTORE, 469, 470, 723 MPI_MODE_NOSUCCEED, 469, 470, 723 MPI_MODE_RDONLY, 514, 515, 520, 723 MPI_MODE_RDWR, 514, 515, 723 MPI_MODE_SEQUENTIAL, 514, 515, 517, 518, 525, 530, 535, 547, 570, 723, 873 MPI_MODE_UNIQUE_OPEN, 514, 515, 723 MPI_MODE_WRONLY, 514, 515, 723 MPI_NO_OP, 445-447, 718 MPI_NULL_COPY_FN, 722 MPI_NULL_DELETE_FN, 722 MPI_OFFSET, 31, 189, 715, 716, 868, 870 MPI_OFFSET_KIND, 16, <u>17</u>, 31, 571, 671, 714 MPI_OP_NULL, 198, 719 MPI_ORDER_C, 15, 104, 105, 107, 108, 724

Unofficial Draft for Comment Only

MPI_ORDER_FORTRAN, 15, 104, 105, 107, 108, 724 MPI_PACKED, 29, 30, 38, 141, 143, 147, 561, 633, 709, 715, 716 MPI_PROC_NULL, 28, 76, 77, 89, 157, 159, 161, 163, 171, 173, 188, 243, 324, 328, 351, 352, 429, 436, 713, 860, 871, 873 MPI_PROD, 188, 189, 718 MPI_QUAD, 863 MPI_REAL, 29, 38, 189, 560, 662, 663, 669, 716 MPI_REAL16, 189, 716 MPI_REAL2, 29, 189, 716 MPI_REAL4, 29, 189, 662, 667, 716 MPI_REAL8, 29, 189, 662, 716, 868 MPI_REPLACE, 443-447, 488, 718, 869, 873 MPI_REQUEST_NULL, 58-61, 63-65, 67, 68, 496, 719 MPI_ROOT, 157, 713 MPI_SEEK_CUR, 541, 548, 724 MPI_SEEK_END, 541, 548, 724 MPI_SEEK_SET, 541, 548, 724 MPI_SHORT, 30, 188, 715 MPI_SHORT_INT, 192, 717 MPI_SIGNED_CHAR, 30, 188, 190, 191, 715, 870 MPI_SIMILAR, 244, 252, 270, 718 MPI_SOURCE, 34, 210, 448, 714 MPI_STATUS, 22, 36, 37, 58, 861 MPI_STATUS_IGNORE, 10, 16, 36, 37, 495, 530, 648, 671, 698, 699, 709, 725, 857, 864 MPI_STATUS_SIZE, 15, 34, 650, 714 MPI_STATUSES_IGNORE, 15, 16, 36, 37, 495, 497, 671, 698, 699, 725, 857 MPI_SUBARRAYS_SUPPORTED, 16, 643-645, 648, 650-653, 657-660, 673, 674, 713, 864 MPI_SUBVERSION, 350, 725 MPI_SUCCESS, 18, 20, 58, 66, 68, 280, 281, 283-286, 289, 290, 365, 366, 370, 371, 396, 566, 590, 609–611, 618, 623, 711 MPI_SUM, 188, 189, 708, 718 MPI_T_BIND_MPI_COMM, 592, 726 MPI_T_BIND_MPI_DATATYPE, 592,

	726	1
MPI_T_	_BIND_MPI_ERRHANDLER,	2
	592, 726	3
MPI T	_BIND_MPI_FILE, 592, 726	4
	BIND_MPI_GROUP, 592, 726	5
	_BIND_MPI_INFO, 592, 726	6
	_BIND_MPI_MESSAGE, 592, 726	7
	_BIND_MPI_OP, 592, 726	8
	_BIND_MPI_REQUEST, 592, 726	9
	_BIND_MPI_WIN, 592, 726	10
	_BIND_NO_OBJECT, 592, 598,	11
	600, 607, 608, 726	12
MPI T	_CVAR_HANDLE_NULL, 600, 725	13
	_ENUM_NULL, 598, 607, 725	14
	_ERR_CANTINIT, 618, 713	15
	_ERR_CVAR_SETNEVER, 601,	16
WII I_I_	618, 713	17
мрі т	_ERR_CVAR_SETNOTNOW, 601,	18
WII I_I_	618, 713	19
MPI T	,	20
		21
MDI T		22
	, ,	23
		24
MP1_1_	713	25
MDI T		26
	$\underline{\text{DRUC}}, \underline{\text{MDRUC}}, \underline{\text{MO}}, \underline{\text{MO}}, \underline{\text{MO}}$	20
MPI_I_	$\underline{\text{DRIC}}, \underline{\text{NOTINITALIZED}}, \underline{\text{OIO}},$	28
MDI T	110	29
MPI_I_	$\underline{\text{Entr}}_{001011111111111111111111111111111111$	30
MDI T	110	31
MPI_T_	$\underline{\text{Entr}}_{00} = 0010101010000000000000000000000000$	32
	110	33
MPI_T_	_LIULI VAIL NODIAUDIOI,	34
	005, 010, 010, 115	35
MPI_T_		36
	011, 010, 710	37
		38
MPI_T_	$1 \text{ VAR}_ADD_11\text{ ADD}_5, 003 012,$	39
	120	40
MPI_T_		41
	004, 121	42
MPI_T_		43
		44
MPI_T_	1 VAILODADD_ODIVERUO, 004 ,	45
	121	46
MPI_T_	_PVAR_CLASS_HIGHWATERMARK,	47
	603, 727	48

1 MPI_T_PVAR_CLASS_LEVEL, 603, 727 2 MPI_T_PVAR_CLASS_LOWWATERMARK, 3 604, 727 4 MPI_T_PVAR_CLASS_PERCENTAGE, $\mathbf{5}$ 603, 727 6 MPI_T_PVAR_CLASS_SIZE, 603, 727 7MPI_T_PVAR_CLASS_STATE, 603, 727 8 MPI_T_PVAR_CLASS_TIMER, 604, 727 9 MPI_T_PVAR_HANDLE_NULL, 609, 725 10MPI_T_PVAR_SESSION_NULL, 608, 725 11 MPI_T_SCOPE_ALL, 598 12MPI_T_SCOPE_ALL_EQ, 598 13MPI_T_SCOPE_GLOBAL, 726 14MPI_T_SCOPE_GLOBAL_EQ, 601, 726 15MPI_T_SCOPE_GROUP, 598, 726 16MPI_T_SCOPE_GROUP_EQ, 598, 601, 17726 18 MPI_T_SCOPE_LOCAL, 598, 726 19MPI_T_SCOPE_READONLY, 598, 726 MPI_T_VERBOSITY_MPIDEV_ALL, 2021591, 726 22 MPI_T_VERBOSITY_MPIDEV_BASIC, 23591, 726 24 MPI_T_VERBOSITY_MPIDEV_DETAIL, 25591, 726 26MPI_T_VERBOSITY_TUNER_ALL, 591, 27726 28 MPI_T_VERBOSITY_TUNER_BASIC, 29 591, 726 30MPI_T_VERBOSITY_TUNER_DETAIL, 31 591, 726 32MPI_T_VERBOSITY_USER_ALL, 591, 33726 34MPI_T_VERBOSITY_USER_BASIC, 35591, 726 36 MPI_T_VERBOSITY_USER_DETAIL, 37591.726 38MPI_TAG, 34, 210, 448, 714 39MPI_TAG_UB, 31, 351, 704, 708, 718 40MPI_THREAD_FUNNELED, 506, 724 41MPI_THREAD_MULTIPLE, 506, 507, 42509.724 43MPI_THREAD_SERIALIZED, 506, 724 44MPI_THREAD_SINGLE, 506, 507, 724 45MPI_TYPECLASS_COMPLEX, 668, 724 46MPI_TYPECLASS_INTEGER, 668, 724 47MPI_TYPECLASS_REAL, 668, 724

MPI_UB, 12, 18, 19, 105, 109, 113, 115, 116, 119, 120, 559, 717, 862 MPI_UINT16_T, 30, 188, 715, 868, 870 MPI_UINT32_T, 30, 189, 715, 868, 870 MPI_UINT64_T, 30, 189, 715, 868, 870 MPI_UINT8_T, 30, 188, 715, 868, 870 MPI_UNDEFINED, 35, 36, 63-65, 68, 69, 112, 115, 117, 121, 122, 144, 242, 243, 258, 315, 326, 327, 664, 713, 861, 870 MPI_UNEQUAL, 244, 252, 270, 718 MPI_UNIVERSE_SIZE, 392, 414, 722 MPI_UNSIGNED, 30, 188, 594, 715 MPI_UNSIGNED_CHAR, 30, 188, 190, 191, 715 MPI_UNSIGNED_LONG, 30, 188, 594, 715 MPI_UNSIGNED_LONG_LONG, 30, 188, 594, 715, 870 MPI_UNSIGNED_SHORT, 30, 188, 715 MPI_UNWEIGHTED, 16, 310, 312, 313, 321, 323, 671, 725, 862, 869 MPI_VAL, 13, 695 MPI_VERSION, 350, 725 MPI_WCHAR, 30, 190, 191, 298, 560, 715, 870 MPI_WEIGHTS_EMPTY, 862 MPI_WIN_BASE, 433, 434, 708, 722 MPI_WIN_CREATE_FLAVOR, 434, 722 MPI_WIN_DISP_UNIT, 433, 434, 722 MPI_WIN_FLAVOR_ALLOCATE, 434, 722 MPI_WIN_FLAVOR_CREATE, 434, 722 MPI_WIN_FLAVOR_DYNAMIC, 434, 722 MPI_WIN_FLAVOR_SHARED, 434, 722 MPI_WIN_MODEL, 434, 454, 722 MPI_WIN_NULL, 433, 719 MPI_WIN_SEPARATE, 434, 454, 473, 722 MPI_WIN_SIZE, 433, 434, 722 MPI_WIN_UNIFIED, 434, 454, 474, 483, 722 MPI_WTIME_IS_GLOBAL, 351, 352, 372, 704, 718

MPI Declarations Index

This index refers to declarations needed in C/C++, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI::Aint, 16, <u>16</u>, 21, 93, <u>93</u>, 96, 99, 102, 111, 115, 116, 126, 148, 149, 423, 425, 427, 430, 436, 438, 442, 444, 446-450, 452, 559, 563, 619-622, 703, 703, 704, 727 MPI::Cartcomm, 305, 630, 636, 728 MPI::Comm, 28, 244, 250-252, 254, 257, 260, 269–273, 279, 282–284, 630, 636, 637, 728 MPI::Datatype, 21, <u>93</u>, 630, 728 MPI::Distgraphcomm, 636, 728, 869 MPI::Errhandler, <u>358</u>, 359–364, 625, 626, 630, 696, 728 MPI:: Exception, 20, 24, 630, 638, 728 MPI::File, 362, 363, 371, 513, 516-522, 524, 526, 530–538, 540, 541, 543-548, 550-556, 559, 568, 569, 630, 696, 728 MPI::Graphcomm, 307, 630, 636, 728 MPI::Grequest, 494, 494, 630, 728 MPI::Group, 242, 242, 243-249, 254, 271, 434, 459, 461, 519, 630, 696, 728 MPI::Info, 353, 383, <u>383</u>, 384–388, 392, 395, 398, 405–408, 410, 513, 516, 521, 522, 524, 630, 696, 728 MPI::Intercomm, 630, 636, 728 MPI::Intracomm, 630, 636, 728 MPI::Message, 696 MPI::Offset, 17, <u>17</u>, 21, 517–519, 524, 526, 530-534, 540, 541, 547, 548, 550, 551, 564, <u>571</u>, 727 MPI::Op, 186, <u>195</u>, 198, 200-204, 206, 207, 223-228, 442, 444, 446, 450, 452, 630, 696, 728 MPI::Prequest, 81, 630, 728

13MPI::Request, 54–57, <u>58</u>, 59, 60, 63–68, 70, 1478, 81-85, 494, 497, 533, 534, 538, 15540, 544, 545, 630, 696, 728 16MPI::Status, 32, 35, 58, 59, 63–68, 70–72, 1775, 76, 77, 79, 86, 88, 121, 495, 18501, 502, 530-532, 535-538, 543, 19546, 551–556, 630, 631, 698, 699, 20727 21MPI::Win, 285, 287, 288, 298, 361, 362, 22 $370, \underline{423}, \underline{425}, \underline{427}, \underline{430}, 432, 434,$ 23436, 438, 442, 444, 446-450, 452, 24 458-462, 464, 465, 467, 468, 630, 25696, 728 26MPI_Aint, 16, <u>16</u>, 17, 20, 31, 93, <u>93</u>, 96, 2799, 102, 111, 115, 116, 126, 148, 28149, 423, 425, 427, 430, 436, 438, 29442, 444, 446–450, 452, 559, 563, 30 619-622, 671, 703, <u>703</u>, 704, 705, 31 727 32 MPI_Comm, 13, 28, 244, 250–252, 254, 33 257, 260, 269–273, 279, 282–284, 34 717, 719, 727, 728 35 MPI_Count, 17, 31, 727, 861 36 MPI_Datatype, 93, 681, 715–717, 719, 727, 37 728 38 MPI_Errhandler, 358, 359-364, 625, 626, 39 696, 714, 719, 727, 728 40 MPI_F08_status, 699, 725, 727, 864 41 MPI_File, 362, 363, 371, 513, 516–522, 42524, 526, 530-538, 540, 541, 43543-548, 550-556, 559, 568, 569, 44696, 719, 727, 728 45MPI_Fint, 695, 725, 727, 870 46MPI_Group, 242, 242, 243-249, 254, 271, 47434, 459, 461, 519, 696, 719, 727, 48

10

11

1	728
2	MPI_Info, 353, 383, <u>383</u> , <u>384</u> –388, 392,
3	395, 398, 405-408, 410, 513, 516,
4	521, 522, 524, 696, 719, 727, 728,
5	873
6	MPI_Message, 696, 719, 727, 860
7	MPI_Offset, 17, <u>17</u> , 20, 31, 517–519, 524,
8	526, 530-534, 540, 541, 547, 548,
9	$550, 551, 564, 571, \underline{571}, 694, 727$
10	MPI_Op, 186, <u>195</u> , 198, 200–204, 206, 207,
11	223-228,442,444,446,450,452,
12	696,718,719,727,728
13	MPI_Request, 54–57, <u>58</u> , 59, 60, 63–68, 70,
14	78, 81 - 85, 494, 497, 533, 534, 538,
15	540, 544, 545, 696, 719, 727, 728
16	MPI_Status, <u>32</u> , 34, 35, 37, 58, 59, 63–68,
17	$70-72, 75, 76, \underline{77}, 79, 86, 88, 121,$
18	495, 501, 502, 530-532, 535-538,
19	543, 546, 551-556, 647, 698-701,
20	725,727,728,864
21	MPI_T_cvar_handle, 599
22	MPI_T_enum, 595
23	MPI_T_pvar_handle, 608
24	MPI_T_pvar_session, 607
25	MPI_Win, 285, 287, 288, 298, 361, 362,
26	$370, \underline{423}, \underline{425}, \underline{427}, \underline{430}, 432, 434,$
27	436, 438, 442, 444, 446-450, 452,
28	458-462, 464, 465, 467, 468, 696,
29	719, 727, 728
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. C++ names for these typedefs and Fortran example prototypes are given near the text of the C name.

	15
MPI_Comm_copy_attr_function, 18, 19,	16
279,645,721,729	17
MPI_Comm_delete_attr_function, 19, 279,	18
721, 729	19
MPI_Comm_errhandler_fn, 626, 869	20
$MPI_Comm_errhandler_function, 19, 358,$	21
626, 729, 869	22
MPI_Copy_function, 19, 622, 722, 735	23
$MPI_Datarep_conversion_function, 564,$	24
729	25
MPI_Datarep_extent_function, 563, 729	26
MPI_Delete_function, 19, 623, 722, 735	27
MPI_File_errhandler_fn, 626, 869	28
MPI_File_errhandler_function, 362, 626,	29
729,869	30
MPI_Grequest_cancel_function, 496, 729	31
MPI_Grequest_free_function, 495, 729	32
MPI_Grequest_query_function, 495, 729	33
MPI_Handler_function, 19, 625, 735, 862	34
MPI_Type_copy_attr_function, 289, 721,	35
729	
MPI_Type_delete_attr_function, 289, 721,	36 37
729,864	
MPI_User_function, 195, 199, 728	38
MPI_Win_copy_attr_function, 285, 721,	39 40
729	
MPI_Win_delete_attr_function, 285, 721,	41 42
729	
MPI_Win_errhandler_fn, 626, 869	43
MPI_Win_errhandler_function, 361, 626,	44
729,869	45
	46
	47
	48

MPI Function Index

10

The underlined page numbers refer to the function definitions.

 11 MPI_ABORT, 196, 357, 374, 377, 416, 12594, 695, 873 13MPI_ACCUMULATE, 421, 422, 435, 436, 14442, 443, 445, 451, 455, 480, 487, 15488, 869, 873 16MPI_ADD_ERROR_CLASS, 368, 368 17MPI_ADD_ERROR_CODE, 369 18MPI_ADD_ERROR_STRING, 369, 370 19MPI_ADDRESS, 19, 111, 621, 654, 656, 20703, 704, 862 21MPI_ALLGATHER, 151, 155, 156, 176, 22 177, 179, 180, 218 23MPI_ALLGATHERV, 151, 155, 156, 178, 24 <u>178</u>, 179, 219 25MPI_ALLOC_MEM, 353, 354, 355, 366, 26425-429, 431, 437, 466, 653, 654, 27656-659, 671, 865 28 MPI_ALLOC_MEM_CPTR, 354, 653 29MPI_ALLREDUCE, 151, 154–156, 188, 30 195, 200, <u>200</u>, 225, 871 31MPI_ALLTOALL, 151, 155, 156, 180, 180, 32 181, 183, 220, 868 33MPI_ALLTOALLV, 151, 155, 156, <u>182</u>, 34183, 185, 221, 868 35MPI_ALLTOALLW, 151, 155, 156, <u>184</u>, 36 185, 223, 868 37 MPI_ATTR_DELETE, 19, 284, 292, 623, 38624 39MPI_ATTR_GET, 19, 284, 292, <u>624</u>, 704, 40705 41MPI_ATTR_PUT, 19, 283, 292, <u>624</u>, 704, 42705, 708 43MPI_BARRIER, 151, 155, 156, 157, 158, 44212, 477, 478, 573 45MPI_BCAST, 151, 155, 156, 158, 158, 159, 46187, 212, 631 47MPI_Bcast, 232 48

MPI_BSEND, <u>43</u>, 52, 353, 375 MPI_BSEND_INIT, 81, 85 MPI_BUFFER_ATTACH, 22, 49, 59 MPI_BUFFER_DETACH, <u>50</u>, 375, 864 MPI_CANCEL, 46, 59, 71, 78, 78, 79, 80, 210, 448, 493, 497, 498 MPI_CART_COORDS, 303, 304, 318, 318, 872 MPI_CART_CREATE, 269, 302-304, 304, 305-307, 317, 325, 326, 328, 672, 871 MPI_CART_GET, 303, 304, 317, 317, 872 MPI_CART_MAP, 304, 326, 326, 860 MPI_CART_RANK, 303, 304, 317, 318, 872 MPI_CART_SHIFT, 303, 304, 323, 324, 324, 328, 872 MPI_CART_SUB, 303, 304, 325, 325, 326, 327, 872 MPI_CARTDIM_GET, 303, 304, 316, 317, 872 MPI_CLOSE_PORT, 405, 406, 409 MPI_COMM_ACCEPT, 404, 405, 406, 407, 408, 414, 416 MPI_COMM_C2F, 695 MPI_COMM_CALL_ERRHANDLER, 370, 371 MPI_COMM_CLONE, 637 MPI_COMM_COMPARE, 251, 270 MPI_COMM_CONNECT, 366, 407, 408, 414, 416 MPI_COMM_CREATE, 249, 254, 254, 255-258, 303, 869 MPI_COMM_CREATE_ERRHANDLER, 19, 357, 358, 358, 360, 625, 733, 864

MPI_Comm_create_errhandler, 731

MPI_COMM_CREATE_GROUP, 862 MPI_COMM_CREATE_KEYVAL, 19, 277, <u>278</u>, 281, 292, 622, 703, 704, 732, 863, 871 MPI_Comm_create_keyval, 729 MPI_COMM_DELETE_ATTR, 19, 277, 281-283, 284, 292, 624 MPI_COMM_DISCONNECT, 292, 397, 416, 416, 417 MPI_COMM_DUP, 244, 249, 252, 253, 255, 260, 271, 273, 277, 280, 284, 292, 300, 622, 862 MPI_COMM_DUP_FN, 19, 280, 280, 281, 282, 654, 721, 863 MPI_COMM_DUP_INFO, 862 MPI_COMM_F2C, 695 MPI_COMM_FREE, 250, 253, 260, 260, 271, 273, 281, 282, 284, 292, 378, 397, 416, 417, 623, 633 MPI_COMM_FREE_KEYVAL, 19, 277, <u>282</u>, 292, 623 MPI_COMM_GET_ATTR, 19, 277, 283, 283, 292, 351, 624, 705, 708 MPI_COMM_GET_ERRHANDLER, 19, 357, 360, 626, 872 MPI_COMM_GET_INFO, 862 MPI_COMM_GET_NAME, 295, 295, 296, 297, 871 MPI_COMM_GET_PARENT, 296, 393, 396, 397 MPI_COMM_GROUP, 14, 242, 244, 244, 249-251, 270, 358, 872 MPI_COMM_IDUP, 860 MPI_COMM_JOIN, <u>417</u>, 418, 419 MPI_COMM_KEYVAL_CREATE, 654 MPI_COMM_NULL_COPY_FN, 19, 280, 280, 281, 282, 703, 721, 863 MPI_COMM_NULL_DELETE_FN, 19, 281, 281, 282, 721 MPI_COMM_RANK, 250, 251, 270, 654 MPI_COMM_REMOTE_GROUP, 271 MPI_COMM_REMOTE_SIZE, 270, 271 MPI_COMM_SET_ATTR, 19, 277, 281, <u>282</u>, 292, 623, 705, 708 MPI_COMM_SET_ERRHANDLER, 19, 357, 359, 625 MPI_COMM_SET_INFO, 862 MPI_COMM_SET_NAME, 294, 295

MPI_COMM_SIZE, 22, 250, 251, 270 1 MPI_COMM_SPAWN, 380, 390-392, 392, $\mathbf{2}$ 3 393, 395–397, 399–401, 414, 415 MPI_COMM_SPAWN_MULTIPLE, 380, 4 390, 391, 397, <u>398</u>, 399, 400, 415 5MPI_COMM_SPLIT, 254, 255, 257, 257, 6 258, 259, 300, 303-305, 307, 325-327, 869 MPI_COMM_SPLIT_TYPE, 862 9 MPI_COMM_TEST_INTER, 269, 269 10 MPI_COMPARE_AND_SWAP, 422, 435, 11 12447, 486 13 MPI_Count, 861 MPI_DIMS_CREATE, 303, 305, 305, 306 1415MPI_DIST_GRAPH_CREATE, 302, 303, 16308, <u>311</u>, 312–314, 323, 328, 869 17 MPI_Dist_graph_create, 312 MPI_DIST_GRAPH_CREATE_ADJACENT, 18 302, 303, 308, <u>309</u>, 310, 314, 323, 1920328, 861, 869 MPI_DIST_GRAPH_NEIGHBOR_COUNT, 21323 22MPI_DIST_GRAPH_NEIGHBORS, 304, 23321, 322, 323, 328, 861, 869 24 25MPI_DIST_GRAPH_NEIGHBORS_COUNT, 26304, 321, 321, 323, 859, 869 MPI_DIST_NEIGHBORS, 303 27MPI_DIST_NEIGHBORS_COUNT, 303 28MPI_DUP_FN, 19, 280, 623 2930 MPI_ERRHANDLER_C2F, 696 31 MPI_ERRHANDLER_CREATE, 19, 358, <u>625</u>, 862, 864 32 33 MPI_ERRHANDLER_F2C, 696 34 MPI_ERRHANDLER_FREE, 357, 364, 872 35 MPI_ERRHANDLER_GET, 19, 358, 360, 36 37 626, 862, 872 MPI_ERRHANDLER_SET, 19, 360, 625, 38 39 862 MPI_ERROR_CLASS, 365, 365, 367 40 MPI_ERROR_STRING, <u>364</u>, 365, 367, 370 4142MPI_EXSCAN, 152, 155, 188, 207, 207, 43228, 869 MPI_F_SYNC_REG, 111, 643, 660, 661, 44662, 682, 684-687, 865 4546MPI_FETCH_AND_OP, 422, 435, 443, 47445, 446 48 MPI_FILE_C2F, 696

Unofficial Draft for Comment Only

1	MPI_FILE_CALL_ERRHANDLER, 371,
2	371
3	MPI_FILE_CLOSE, 417, 513, 514, 516,
4	<u>516</u>
5	MPI_FILE_CREATE_ERRHANDLER,
6	357, 358, <u>362</u> , 363, 733, 864
7	MPI_File_create_errhandler, 731
8	MPI_FILE_DELETE, 515, 516, <u>516</u> , 517,
9	521, 523, 577
10	$MPI_FILE_F2C, 696$
11	MPI_FILE_GET_AMODE, 520, 520
12	MPI_FILE_GET_ATOMICITY, 569, 569
13	MPI_FILE_GET_BYTE_OFFSET, 535,
14	541, 541, 542, 548
15	MPI_FILE_GET_ERRHANDLER, 357,
16	<u>363, 577, 872</u>
17	MPI_FILE_GET_GROUP, 519, 519
18	MPI_FILE_GET_INFO, 522, 523, 873
19	MPI_FILE_GET_POSITION, 541, 541
20	MPI_FILE_GET_POSITION_SHARED,
21	$547, 548, \underline{548}, 570$
22	MPI_FILE_GET_SIZE, 519, <u>519</u> , 572
23	MPI_FILE_GET_TYPE_EXTENT, 558,
24	559, 559, 566
25	MPI_FILE_GET_VIEW, <u>526</u> , 527
26	MPI_FILE_IREAD, 527, <u>538</u> , 539, 549,
27	567
28	MPI_FILE_IREAD_AT, 527, <u>533</u> , 534
29	MPI_FILE_IREAD_SHARED, 527, 544,
30	<u>544</u>
31	MPI_FILE_IWRITE, 527, 540, <u>540</u>
32	MPI_FILE_IWRITE_AT, 527, 534, <u>534</u>
33	MPI_FILE_IWRITE_SHARED, 527, 545,
34	$\underline{545}$
35	MPI_FILE_OPEN, 367 , 505, 513, <u>513</u> ,
36	514, 515, 521- $523, 525, 542, 571,$
37	572,577,578
38	MPI_FILE_PREALLOCATE, 517, 518,
39	$\underline{518},567,572$
40	MPI_FILE_READ, 527, 528, 535, <u>535</u> ,
41	537,539,571,572
42	MPI_FILE_READ_ALL, 527, <u>536</u> , 537,
43	549,550
44	$\mathbf{MPI_FILE_READ_ALL_BEGIN, 527},$
45	$549,550,\underline{552},567,687$
46	MPI_FILE_READ_ALL_END, 527, 549,
47	550, 553, 567, 687
48	MPI_FILE_READ_AT, 527, <u>530</u> , 531, 534

MPI_FILE_READ_AT_ALL, 527, 531, 531MPI_FILE_READ_AT_ALL_BEGIN, 527, 550, 687 MPI_FILE_READ_AT_ALL_END, 527, <u>551</u>, 687 MPI_FILE_READ_ORDERED, 527, 546, 546 MPI_FILE_READ_ORDERED_BEGIN, 527, <u>554</u>, 687 MPI_FILE_READ_ORDERED_END, 527, 555, 687 MPI_FILE_READ_SHARED, 527, 543, <u>543</u>, 544, 546 MPI_FILE_SEEK, 540, 541 MPI_FILE_SEEK_SHARED, 547, 547, 548, 570 MPI_FILE_SET_ATOMICITY, 515, 568, 568, 569 MPI_FILE_SET_ERRHANDLER, 357, <u>363</u>, 577 MPI_FILE_SET_INFO, 521, 522, 523, 873 MPI_FILE_SET_SIZE, 517, 517, 518, 567, 570, 572 MPI_FILE_SET_VIEW, 106, 367, 514, 521-524, 524, 525, 526, 541, 548, 557, 563, 571, 578, 873 MPI_FILE_SYNC, 516, 528, 567, 568, 569, 570, 574 MPI_FILE_WRITE, 527, 528, 537, 537, 538, 540, 571 MPI_FILE_WRITE_ALL, 527, 538, 538 MPI_FILE_WRITE_ALL_BEGIN, 527, 553, 673, 687 MPI_FILE_WRITE_ALL_END, 527, 554, 687 MPI_FILE_WRITE_AT, 527, 528, 532, 532, 533, 534 MPI_FILE_WRITE_AT_ALL, 527, 532, 533MPI_FILE_WRITE_AT_ALL_BEGIN, 527, 551, 651, 687 MPI_FILE_WRITE_AT_ALL_END, 527, 552, 687 MPI_FILE_WRITE_ORDERED, 527, 545, 546, 547

MPI_FILE_WRITE_ORDERED_BEGIN,

$527, \underline{555}, 687$	MPI_GREQUEST_START, 494
MPI_FILE_WRITE_ORDERED_END,	870
527, 556, 687	MPI_Grequest_start, 731
MPI_FILE_WRITE_SHARED, 527, 528,	MPI_GROUP_C2F, 696
543, 544, 545, 547	MPI_GROUP_COMPARE, 243
MPI_FINALIZE, 15, 25, 350, 351, 374,	MPI_GROUP_DIFFERENCE,
374, 375-379, 416, 417, 504, 514,	MPI_GROUP_EXCL, 247, 247,
$\overline{590}$, 599, 612, 614, 695, 699, 861,	MPI_GROUP_F2C, <u>696</u>
863, 872	MPI_GROUP_FREE, <u>249</u> , 250,
MPI_FINALIZED, 21, 373, 376, 378, 379,	872
379,695	MPI_GROUP_INCL, <u>246</u> , 247, 2
MPI_FREE_MEM, <u>354</u> , 355, 366, 426,	MPI_GROUP_INTERSECTION
427, 654	MPI_GROUP_RANGE_EXCL,
MPI_GATHER, 151, 154–156, <u>160</u> , 162,	MPI_GROUP_RANGE_INCL,
163, 171, 177, 187, 214	MPI_GROUP_RANK, 242, 251
MPI_GATHERV, 151, 155, 156, 162, <u>162</u> ,	MPI_GROUP_SIZE, <u>242</u> , 250
163, 164, 172, 178, 179, 215	MPI_GROUP_TRANSLATE_R
MPI_GET, 421, 435, 436, <u>438</u> , 439, 445,	$243, \underline{243}, 871$
450, 455, 477, 479, 480, 685, 873	MPI_GROUP_UNION, <u>245</u>
MPI_GET_ACCUMULATE, 422, 435,	MPI_IALLGATHER, 151, 155,
443, <u>444</u> , 445, 453, 480, 486, 487	MPI_IALLGATHERV, 151, 155
MPI_GET_ADDRESS, 19, 93, 110, 111,	MPI_IALLREDUCE, 151, 155,
111, 112, 123, 430, 621, 654, 656,	MPI_IALLTOALL, 151, 155, 15
676, 681, 701-703	MPI_IALLTOALLV, 151, 155, 1
MPI_GET_COUNT, 35, <u>35</u> , 36, 58, 121,	MPI_IALLTOALLW, 151, 155, 1
122, 448, 501, 502, 530, 861, 870	MPI_IBARRIER, 151, 155, 156,
MPI_GET_ELEMENTS, 58, 121, <u>121</u> ,	212
122,501,502,530,861	MPI_Ibarrier, 232
MPI_GET_ELEMENTS_X, 58, 121, <u>121</u> ,	MPI_IBCAST, 151, 155, 156, <u>21</u>
122, 501, 502, 861	MPI_IBSEND, <u>55</u> , 59, 85
MPI_GET_LIBRARY_VERSION, 350,	MPI_IEXSCAN, 152, 155, <u>228</u>
$\underline{350},351,373,376,860$	MPI_IGATHER, 151, 155, 156,
MPI_GET_PROCESSOR_NAME, <u>352</u> ,	MPI_IGATHERV, 151, 155, 156
353,872	MPI_IMPROBE, 71, 74, 75, <u>75</u> ,
MPI_GET_VERSION, 350, <u>350</u> , 351, 373,	859,864
376,659	MPI_IMRECV, 74–76, 78, <u>78</u>
MPI_GRAPH_CREATE, 302, 303, <u>306</u> ,	MPI_INEIGHBOR_ALLGATH
307, 308, 313, 316, 319, 320, 327,	338,861
328,871	MPI_INEIGHBOR_ALLGATH
MPI_GRAPH_GET, 303, 316, <u>316</u>	$\underline{339}, 861$
MPI_GRAPH_MAP, 304, 327, <u>327</u>	MPI_INEIGHBOR_ALLTOALL
MPI_GRAPH_NEIGHBORS, 303, 304,	861
$319, \underline{319}, 320, 328, 869$	MPI_INEIGHBOR_ALLTOALL
MPI_GRAPH_NEIGHBORS_COUNT,	341, 861
$303, 319, \underline{319}, 320, 869$	MPI_INEIGHBOR_ALLTOALI
MPI_GRAPHDIMS_GET, 303, <u>315</u> , 316	$\underline{342}, 861$
MPI_GREQUEST_COMPLETE, 494–497,	MPI_INFO_C2F, <u>696</u>
497, 498	MPI_INFO_CREATE, 384, <u>384</u>

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
$\frac{46}{47}$

Unofficial Draft for Comment Only

1	MPI_INFO_DELETE, 366, 385, <u>385</u> , 387
2	MPI_INFO_DUP, <u>387</u> , <u>388</u>
3	MPI_INFO_F2C, $\underline{696}$
4	MPI_INFO_FREE, <u>388</u> , 522
5	MPI_INFO_GET, 383, 385, 873
6	MPI_INFO_GET_NKEYS, 383, <u>386</u> , 387,
7	
	873
8	MPI_INFO_GET_NTHKEY, 383, <u>387</u> ,
9	873
10	MPI_INFO_GET_VALUELEN, <u>383</u> , <u>386</u> ,
11	873
12	MPI_INFO_SET, 384, <u>384</u> , 386, 387
13	MPI_INIT, 15, 25, 241, 350, 351, 373, <u>373</u> ,
14	376-379, 393-395, 397, 414, 415,
15	505-508, 587, 590, 593-595, 599,
16	612, 694, 695, 699, 861, 863, 867,
17	869
18	MPI_INIT_THREAD, 241, 373, 378, 506,
19	<u>506</u> , 507, 508, 593, 694, 860, 863,
20	$\frac{500}{869}$, 507, 508, 595, 094, 800, 805, 869
21	
21	MPI_INITIALIZED, 373, 376, 377, <u>377</u> ,
	379, 508, 695
23	MPI_INTERCOMM_CREATE, 252, 271,
24	$\underline{272}, 273, 862$
25	MPI_INTERCOMM_MERGE, 268,
26	271-273, 273, 864
27	MPI_IPROBE, 36, 71, <u>71</u> , 72–76, 78, 504,
28	859,864
29	MPI_IRECV, <u>57</u> , 78, 674–676, 679, 680
30	MPI_IREDUCE, 151, 155, 156, <u>223</u> , 224
31	MPI_IREDUCE_SCATTER, 151, 155,
32	156, 226
33	MPI_IREDUCE_SCATTER_BLOCK,
34	$151, 155, 156, \underline{225}$
35	MPI_IRSEND, 56
36	MPI_IS_THREAD_MAIN, 506, 508
37	MPI_ISCAN, 152, 155, 227
38	MPI_ISCATTER, 151, 155, 156, <u>215</u>
39	
40	MPI_ISCATTERV, 151, 155, 156, <u>216</u>
	MPI_ISEND, 11, <u>54</u> , 84, 374, 654, 655,
41	673, 675, 680, 681
42	MPI_ISSEND, 55
43	MPI_KEYVAL_CREATE, 19, 279, 281,
44	$\underline{622}, 624, 735$
45	MPI_KEYVAL_FREE, 19, 282, 292, <u>623</u>
46	MPI_LOOKUP_NAME, 366, 404, 409,
47	410, 411
48	MPI_MESSAGE_C2F, <u>696</u> , 860
	,,

MPI_MESSAGE_F2C, <u>696</u>, 860 MPI_MPROBE, 71, 74-76, 76, 78, 859, 864 MPI_MRECV, 74-77, 77, 78, 860 MPI_NEIGHBOR_ALLGATHER, 304, <u>329</u>, 330, 332, 338, 861 MPI_NEIGHBOR_ALLGATHERV, 304, <u>331</u>, 339, 861 MPI_NEIGHBOR_ALLTOALL, 304, 332, 333, 340, 861 MPI_NEIGHBOR_ALLTOALLV, 304, 334, 341, 861 MPI_NEIGHBOR_ALLTOALLW, 304, 335, <u>336</u>, 343, 861 MPI_NULL_COPY_FN, 18, 19, 280, 623, 645 MPI_NULL_DELETE_FN, 19, 281, 623 MPI_OP_C2F, 696 MPI_OP_COMMUTATIVE, 202, 868 MPI_OP_CREATE, 195, <u>195</u>, 197, 654, 732, 864 MPI_Op_create, 729 MPI_OP_F2C, <u>696</u> MPI_OP_FREE, 198 MPI_OPEN_PORT, 404, 405, 405, 407-409, 411 MPI_PACK, 52, <u>141</u>, 144, 147, 561, 564, 565MPI_PACK_EXTERNAL, 8, 147, <u>148</u>, 666, 871 MPI_PACK_EXTERNAL_SIZE, 149 MPI_PACK_SIZE, 52, 144, 144, 861 MPI_PCONTROL, 585, 586, 586, 587 MPI_PROBE, 33, 36, 37, 71, 72, 72, 73-76, 78, 504, 859, 864 MPI_PUBLISH_NAME, 404, 408, 409-411 MPI_PUT, 421, 435, 436, <u>436</u>, 439, 443, 449, 455, 460, 466, 470, 473, 478, 479, 673, 685, 873 MPI_QUERY_THREAD, 508, 509 MPI_RACCUMULATE, 422, 435, 443, 445, 450, 451 MPI_RECV, 28, 32, 34, 36, 37, 72, 74, 75, 92, 120, 142, 143, 152, 161, 502, 573, 612, 631, 681, 684-686 MPI_Recv, 233 MPI_RECV_INIT, <u>83</u>, 84 MPI_REDUCE, 151, 155, 156, 186, <u>186</u>,

Unofficial Draft for Comment Only

187, 188, 195-197, 200, 203, 205-207, 224, 443, 445-447, 869 MPI_REDUCE_LOCAL, 187, 188, 195, 201, 864, 868 MPI_REDUCE_SCATTER, 151, 155, 156, 188, 195, 204, <u>204</u>, 205, 227 MPI_REDUCE_SCATTER_BLOCK, 151, 155, 156, 188, 195, 203, <u>203</u>, 204, 226.868 MPI_REGISTER_DATAREP, 367, 561, 563, 564, 566, 578, 654, 734 MPI_Register_datarep, 731 MPI_Request, 674 MPI_REQUEST_C2F, 696 MPI_REQUEST_F2C, 696 MPI_REQUEST_FREE, 22, 60, 61, 79, 85, 210, 374, 375, 448, 496, 498, 868 MPI_REQUEST_GET_STATUS, 37, 70, 71, 495, 868 MPI_RGET, 421, 435, <u>449</u>, 450 MPI_RGET_ACCUMULATE, 422, 435, 443, 445, 452, 453 MPI_RPUT, 421, 435, 448, 449 MPI_RSEND, 44 MPI_RSEND_INIT, 83 MPI_SCAN, 152, 155, 188, 195, 206, 206, 208, 227 MPI_SCATTER, 151, 155, 156, 170, 171-173, 203, 216 MPI_SCATTERV, 151, 155, 156, 172, 172, 173, 174, 205, 217 MPI_SEND, 27, 28, 29, 36, 39, 92, 119, 141, 343, 514, 573, 587, 631, 652-655, 672, 681, 682, 684, 685 MPI_Send, 233 MPI_SEND_INIT, <u>81</u>, 84 MPI_SENDRECV, 86, 323 MPI_SENDRECV_REPLACE, 88 MPI_SIZEOF, 644, 646, 667, 668 MPI_SSEND, 44 MPI_SSEND_INIT, 82 MPI_START, 84, 84, 85 MPI_STARTALL, 85, 85 MPI_STATUS_C2F, 699 MPI_STATUS_C2F08, 699, 864 MPI_STATUS_F082C, 699, 864 MPI_STATUS_F082F, 701, 864 MPI_STATUS_F2C, 698

MPI_STATUS_F2F08, 700, 864 1 MPI_STATUS_SET_CANCELLED, 502 $\mathbf{2}$ 3 MPI_STATUS_SET_ELEMENTS, 501, 4 502MPI_STATUS_SET_ELEMENTS_X, 501, $\mathbf{5}$ 6 502,861 $\overline{7}$ MPI_T_CATEGORY_CHANGED, 617 MPI_T_CATEGORY_GET_CATEGORIES, 8 616, 616, 617 9 MPI_T_CATEGORY_GET_CVARS, 616, 10 11 616.617 12MPI_T_CATEGORY_GET_INFO, 615, 13 617 MPI_T_CATEGORY_GET_NUM, 615 1415MPI_T_CATEGORY_GET_PVARS, 616, 16616, 617 17MPI_T_CVAR_GET_INFO, 595, 597, 18 597, 598, 600, 601, 617 MPI_T_CVAR_GET_NUM, 597, 600 19MPI_T_CVAR_HANDLE_ALLOC, 600, 2021<u>600</u>, 601 MPI_T_CVAR_HANDLE_FREE, 600, 600 22MPI_T_CVAR_READ, 601 23MPI_T_CVAR_WRITE, 601 24 MPI_T_ENUM_GET_INFO, 595, 595 2526MPI_T_ENUM_GET_ITEM, 595, 596 MPI_T_ENUMITEM, 618 27MPI_T_FINALIZE, 594, 594 28MPI_T_INIT_THREAD, 593, 593, 594 29MPI_T_PVAR_GET_INFO, 595, 604-606, 30 31 606, 608, 610, 611, 617 MPI_T_PVAR_GET_NUM, 605, 608 32 33 MPI_T_PVAR_HANDLE_ALLOC, 608, 34608, 610 MPI_T_PVAR_HANDLE_FREE, 609, 609 35MPI_T_PVAR_READ, 610, 610, 611 36 MPI_T_PVAR_READRESET, 607, 611, 37 611, 618 3839MPI_T_PVAR_RESET, 611, 611, 618 MPI_T_PVAR_SESSION_CREATE, 607 40 41 MPI_T_PVAR_SESSION_FREE, 607 42MPI_T_PVAR_START, 609, 618 MPI_T_PVAR_STOP, 609, 618 43MPI_T_PVAR_WRITE, 610, 610, 618 44MPI_TEST, 11, 37, 58, <u>59</u>, 60-62, 65, 79, 454685, 374, 375, 497, 498, 528, 529 MPI_TEST_CANCELLED, 58-60, 79, 80, 47495, 502, 530 48

1	MPI_TESTALL, 63, <u>66</u> , 67, 495–497, 500,
2	504
3	MPI_TESTANY, 58, 63, <u>64</u> , 65, 69,
4	495-497, 500, 504
5	MPI_TESTSOME, <u>63</u> , <u>68</u> , <u>69</u> , <u>495–497</u> ,
6	500, 504
7	MPI_TOPO_TEST, 303, 315, <u>315</u>
8	MPI_TYPE_C2F, <u>696</u>
9	MPI_TYPE_COMMIT, <u>117</u> , 118, 696
10	MPI_TYPE_CONTIGUOUS, 12, 93, 93,
11	95, 113, 125, 512, 559
12	MPI_TYPE_CREATE_DARRAY, 12, 36,
13	<u>106</u> , 107, 125
14	MPI_TYPE_CREATE_F90_COMPLEX,
15	,
16	12, 125, 127, 189, 561, 636, 644,
17	$647, \underline{664}, 666$
18	MPI_TYPE_CREATE_F90_INTEGER,
19	12, 125, 127, 189, 561, 636, 644,
	646, <u>664</u> , 666
20	MPI_TYPE_CREATE_F90_REAL, 12,
21	125, 127, 189, 561, 636, 644, 646,
22	<u>663,</u> 664–666, 868
23	MPI_TYPE_CREATE_HINDEXED, 12,
24	19, 93, 99, <u>99</u> , 101, 103, 125, 619
25 26	MPI_TYPE_CREATE_HINDEXED_BLOCK
20	$12, 93, 101, \underline{101}, 125, 862$
21	MPI_TYPE_CREATE_HVECTOR, 12,
28 29	19, 93, 96, <u>96</u> , 125, 619
29 30	MPI_TYPE_CREATE_INDEXED_BLOCK,
31	12, <u>100</u> , 101, 125
32	MPI_TYPE_CREATE_KEYVAL, 277,
33	<u>288</u> , 292, 704, 733, 871
34	MPI_Type_create_keyval, 730
35	MPI_TYPE_CREATE_RESIZED, 18, 19,
36	93, <u>115</u> , 125, 559, 864
37	MPI_TYPE_CREATE_STRUCT, 12, 19,
	93, <u>102</u> , 103, 113, 125, 185, 620
38 39	MPI_TYPE_CREATE_SUBARRAY, 12,
40	15, <u>103</u> , 105, 107, 125
40	MPI_TYPE_DELETE_ATTR, 277, 292,
	<u>292</u> , 864
42	MPI_TYPE_DUP, 12, 119, <u>119</u> , 125, 864
43	MPI_TYPE_DUP_FN, 289, <u>289</u> , 721
44	MPI_TYPE_EXTENT, 19, 114, 115, <u>621</u> ,
45	704, 862
46	MPI_TYPE_F2C, <u>696</u>
47	MPI_TYPE_FREE, <u>118</u> , 127, 290
48	MPI_TYPE_FREE_KEYVAL, 277, <u>290</u> ,

- MPI_TYPE_GET_ATTR, 277, <u>291</u>, 292, 705, 864 MPI_TYPE_GET_CONTENTS, 124, 125, <u>126</u>, 127–129
- MPI_TYPE_GET_ENVELOPE, 124, <u>124</u>, 127, 128, 665
- MPI_TYPE_GET_EXTENT, 19, <u>115</u>, 117, 621, 668, 701, 704
- MPI_TYPE_GET_EXTENT_X, 861
- MPI_TYPE_GET_NAME, 297, 864
- MPI_TYPE_GET_TRUE_EXTENT, 116
- $\begin{array}{c} \mathrm{MPI_TYPE_GET_TRUE_EXTENT_X},\\ 861 \end{array}$
- MPI_TYPE_HINDEXED, 19, 99, 125, 126, <u>620</u>, 704, 862 MPI_TYPE_HVECTOR_10_06_125_126
- MPI_TYPE_HVECTOR, 19, 96, 125, 126, <u>619</u>, 704, 862
- MPI_TYPE_INDEXED, 12, 97, <u>97</u>, 98–100, 125
- MPI_TYPE_LB, 19, 114, 115, 621, 862
- MPI_TYPE_MATCH_SIZE, 644, 646, 668, 668, 864
- MPI_TYPE_NULL_COPY_FN, 289, <u>289</u>, , 721
- MPI_TYPE_NULL_DELETE_FN, <u>289</u>, 721, 864
- MPI_TYPE_SET_ATTR, 277, <u>291</u>, 292, 705, 708, 864
- MPI_TYPE_SET_NAME, 297, 864
- MPI_TYPE_SIZE, 112, <u>112</u>, 587, 861
- MPI_TYPE_SIZE_X, 861
- MPI_TYPE_STRUCT, 18, 19, 102, 113, 125, 126, <u>620</u>, 704, 862
- MPI_TYPE_UB, 19, 114, 115, <u>622</u>, 704, 862
- MPI_TYPE_VECTOR, 12, 94, <u>94</u>, 95, 96, 98, 125
- MPI_UNPACK, <u>142</u>, 143, 147, 565
- MPI_UNPACK_EXTERNAL, 8, 149, 666
- MPI_UNPUBLISH_NAME, 366, 410, <u>410</u>
- $\begin{array}{c} \text{MPI_WAIT, 35, 37, 58, } \underline{58, 59} \underline{62, 64, 65,} \\ & 79, 85, 210, 374, 375, 493, 497, \\ & 498, 504, 528, 529, 549, 567, 569, \\ & 673, 674, 680, 681, 684, 686 \end{array}$
- MPI_Wait, 233
- MPI_WAITALL, 63, 65, <u>65</u>, 66, 67, 210, 448, 495–497, 500, 504, 639

MPI_WAITANY, 46, 58, 63, <u>63</u>, 64, 69, 495-497, 500, 504 MPI_WAITSOME, 63, 67, 68–70, 495–497, 500, 504 MPI_WIN_ALLOC, 428 MPI_WIN_ALLOCATE, 422, 425, 426, 433, 434, 437, 466, 654, 657-659 MPI_WIN_ALLOCATE_CPTR, 426 MPI_WIN_ALLOCATE_SHARED, 422, 427, 427, 429, 434 MPI_WIN_ALLOCATE_SHARED_CPTR, 428 MPI_WIN_ATTACH, 430, 431, 431, 432, 466 MPI_WIN_C2F, 696 MPI_WIN_CALL_ERRHANDLER, 370, 371MPI_WIN_COMPLETE, 433, 455, 460, 460, 461-463, 472, 478, 479 MPI_WIN_CREATE, 422, 423, 425, 426, 428, 430, 431, 433, 434, 470, 505 MPI_WIN_CREATE_DYNAMIC, 422, 430, 430, 431, 433, 434, 437, 471 MPI_WIN_CREATE_ERRHANDLER, 357, 358, 360, 361, 733, 864 MPI_Win_create_errhandler, 731 MPI_WIN_CREATE_KEYVAL, 277, 285, 292, 704, 732, 871 MPI_Win_create_keyval, 730 MPI_WIN_DELETE_ATTR, 277, 288, 292MPI_WIN_DETACH, 430, 432, 432, 433 MPI_WIN_DUP_FN, 285, 285, 721 MPI_WIN_F2C, <u>696</u> MPI_WIN_FENCE, 433, 439, 455, 458, 459, 469, 472, 473, 476, 481, 685 MPI_WIN_FLUSH, 428, 448, 449, 467, 467, 472, 475, 487, 488 MPI_WIN_FLUSH_ALL, 448, 449, 467, 467, 472, 475 MPI_WIN_FLUSH_LOCAL, 448, 467, 468, 472 MPI_WIN_FLUSH_LOCAL_ALL, 448, 468, 468, 472 MPI_WIN_FREE, 286, 417, 432, 433 MPI_WIN_FREE_KEYVAL, 277, 287, 292 MPI_WIN_GET_ATTR, 277, 287, 292, 433, 705, 708

MPI_Win_get_attr, 433, 434 1 MPI_WIN_GET_ERRHANDLER, 357, $\mathbf{2}$ 362, 872 MPI_WIN_GET_GROUP, <u>434</u>, 435 4 MPI_WIN_GET_INFO, 862 5MPI_WIN_GET_NAME, 298 6 MPI_WIN_LOCK, 353, 424, 456, 464, 464, 7 465, 466, 469, 472, 476-478 MPI_WIN_LOCK_ALL, 424, 456, 464, 9 10 464, 465, 469, 472, 478, 487 MPI_WIN_NULL_COPY_FN, 285, 285, 1112721 MPI_WIN_NULL_DELETE_FN, 285, 721 13 MPI_WIN_POST, 433, 455, 460, 461, 461, 1415462, 463, 466, 469, 472, 479, 481 16MPI_WIN_SET_ATTR, 277, 287, 292, 17705, 708 18MPI_Win_set_attr, 433, 434 MPI_WIN_SET_ERRHANDLER, 357, 1920361 21MPI_WIN_SET_INFO, 862 MPI_WIN_SET_NAME, 298 22MPI_WIN_SHARED_QUERY, 427, 428 23MPI_WIN_SHARED_QUERY_CPTR, 24 25429 MPI_WIN_START, 455, 459, 460, 461, 26463, 469, 479, 486 2728MPI_WIN_SYNC, 468, 468, 472, 474, 478, 29487, 488 MPI_WIN_TEST, 462, 462, 463 30 31 MPI_WIN_UNLOCK, 353, 449, 456, 465, 466, 472, 476, 477 32 33 MPI_WIN_UNLOCK_ALL, 449, 456, 465, 34472, 476, 487 MPI_WIN_WAIT, 433, 455, 461, 461, 462, 35 463, 466, 472, 476, 478-480 36 MPI_WTICK, 23, 372, 373 37 MPI_WTIME, 23, 352, 372, <u>372</u>, 373, 587, 3839 604 mpiexec, 374, 378, 379, 380, 507 40 mpirun, 379 41 42PMPI_, 584 43PMPI_SEND, 653 44PMPI_WTICK, 23 45PMPI_WTIME, 23 464748