## MPI: A Message-Passing Interface Standard Version 3.1

ticket0.

Message Passing Interface Forum

September 21, 2012

1	This document describes the Message-Passing Interface (MPI) standard, version 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
6	C and Fortran are defined.
7	Historically, the evolution of the standards is from MPI-1.0 (June 1994) to MPI-1.1
8	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
9	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality,
10	to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
11	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
12	combining the previous documents. Version MPI-2.2 (September 2009) added additional
13	clarifications and seven new routines. This version, MPI-3.0, is an extension of MPI-2.2.
14	
15	Comments. Please send comments on MPI to the MPI Forum as follows:
16	comments. I lease send comments on with to the with Forum as follows.
17	1. Subscribe to http://lists.mpi-forum.org/mailman/listinfo.cgi/mpi-comments
18	
19	2. Send your comment to: mpi-comments@mpi-forum.org, together with the URL of
20	the version of the MPI standard and the page and line numbers on which you are
21	commenting. Only use the official versions.
22	Your comment will be forwarded to MPI Forum committee members for consideration.
23	Messages sent from an unsubscribed e-mail address will not be considered.
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012 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.

ii

Version 3.0: September 21, 2012. 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 contains significant extensions to MPI functionality, including nonblocking collectives, new one-sided communication operations, and Fortran 2008 bindings. 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 document. A few extensions have been added; however all correct MPI-2.1 programs are correct 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 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 do not fit elsewhere, in particular language interoperability.

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 make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes from Version 1.0 are minor. A version of this document with all changes marked is available.

Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since January 1993 to discuss and define a set

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

 $\frac{44}{45}$ 

of library interface standards for message passing. MPIF is not sanctioned or supported by
 any official standards organization.

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.

<sup>6</sup> This is the final report, Version 1.0, of the Message-Passing Interface Forum. This <sup>7</sup> document contains all the technical features proposed for the interface. This copy of the <sup>8</sup> draft was processed by LATEX on May 5, 1994.

# Contents

A	cknov	wledgments	ix								
1	Intr	oduction to MPI	1								
	1.1										
1.2 Background of MPI-1.0											
	1.3										
	1.4	Background of MPI-1.3 and MPI-2.1	3								
	1.5	Background of MPI-2.2.	4								
	1.6	Background of MPI-3.0	4								
	1.7	Who Should Use This Standard?	4								
	1.8	What Platforms Are Targets For Implementation?	$\overline{5}$								
	1.9	What Is Included In The Standard?	5								
		What Is Not Included In The Standard?	6								
		Organization of this Document	6								
			Ŭ								
2	MPI	Terms and Conventions	9								
	2.1	Document Notation	9								
	2.2	Naming Conventions	9								
	2.3	Procedure Specification	10								
	2.4	Semantic Terms	11								
	2.5	Data Types	12								
		2.5.1 Opaque Objects	12								
		2.5.2 Array Arguments	14								
		2.5.3 State	15								
		2.5.4 Named Constants	15								
		2.5.5 Choice	16								
		2.5.6 Addresses	16								
		2.5.7 File Offsets	16								
		2.5.8 Counts	17								
	2.6	Language Binding	17								
		2.6.1 Deprecated and Removed Names and Functions	17								
		2.6.2 Fortran Binding Issues	19								
		2.6.3 C Binding Issues	19								
		2.6.4 Functions and Macros	20								
	2.7	Processes	$\frac{-0}{20}$								
	2.8	Error Handling	$\frac{-0}{20}$								
	2.9	Implementation Issues	$\overline{21}$								

		2.9.1	Independence of Basic Runtime Routines .										•		22
		2.9.2	Interaction with Signals												22
	2.10	Examp	bles												22
	ъ.														
3			Point Communication												23
	3.1		uction												23
	3.2		ng Send and Receive Operations												24
		3.2.1	Blocking Send												24
		3.2.2	Message Data												25
		3.2.3	Message Envelope												27
		3.2.4	Blocking Receive												28
		3.2.5	Return Status												30
		3.2.6	Passing $MPI\_STATUS\_IGNORE$ for Status $% Status$ .	•						•					32
	3.3	Data 7	Type Matching and Data Conversion	•				•		•			•	•	33
		3.3.1	Type Matching Rules										•		33
			Type MPI_CHARACTER										•		34
		3.3.2	Data Conversion												35
	3.4	Comm	unication Modes										•		37
	3.5	Seman	tics of Point-to-Point Communication												40
	3.6		Allocation and Usage												44
		3.6.1	Model Implementation of Buffered Mode												46
	3.7	Nonblo	ocking Communication												47
		3.7.1	Communication Request Objects												48
		3.7.2	Communication Initiation												48
		3.7.3	Communication Completion												52
		3.7.4	Semantics of Nonblocking Communications												56
		3.7.5	Multiple Completions												57
		3.7.6	Non-destructive Test of status												63
	3.8		and Cancel												64
	0.0	3.8.1	Probe												64
		3.8.2	Matching Probe												67
			Matched Receives												
		3.8.3													69 71
	2.0	3.8.4	Cancel												71
	3.9		cent Communication Requests												73 70
			Receive												78
	3.11	Null P	rocesses	·	·	·	•••	•	• •	•	·	• •	•	·	80
4	Dat	atypes													83
	4.1		d Datatypes												83
		4.1.1	Type Constructors with Explicit Addresses												85
		4.1.2	Datatype Constructors												85
		4.1.3	Subarray Datatype Constructor												94
		4.1.4	Distributed Array Datatype Constructor												96
		4.1.5	Address and Size Functions												101
		4.1.6	Lower-Bound and Upper-Bound Markers												101
															105
		4.1.7	Extent and Bounds of Datatypes												
		4.1.8 4 1 9	True Extent of Datatypes	·	•	·	• •	•	• •	•	·	• •	•	·	$107 \\ 109$
		414													1119

		4.1.10	Duplicating a Datatype	 	 . 110
		4.1.11	Use of General Datatypes in Communication	 	 . 111
		4.1.12	Correct Use of Addresses	 	 . 114
		4.1.13	Decoding a Datatype	 	 . 115
		4.1.14	Examples	 	 . 121
	4.2		und Unpack		
	4.3		ical MPI_PACK and MPI_UNPACK		
5	Coll	lective	Communication		141
	5.1		uction and Overview	 	 . 141
	5.2	Comm	unicator Argument	 	 . 144
		5.2.1	Specifics for Intracommunicator Collective Operations	 	 . 144
		5.2.2	Applying Collective Operations to Intercommunicators		
		5.2.3	Specifics for Intercommunicator Collective Operations		
	5.3	Barrie	r Synchronization		
	5.4		cast		
		5.4.1	Example using MPI_BCAST		
	5.5	Gather	r		
		5.5.1	Examples using MPI_GATHER, MPI_GATHERV		
	5.6	Scatte	$\mathbf{r}$		
		5.6.1	Examples using MPI_SCATTER, MPI_SCATTERV		
	5.7		r-to-all		
		5.7.1	Example using MPI_ALLGATHER		
	5.8		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		
		5.9.7	Process-Local Reduction		
	5 10		e-Scatter		
	0.10		MPI_REDUCE_SCATTER_BLOCK		
			MPI_REDUCE_SCATTER		
	5.11				
	0.11		Inclusive Scan		
			Exclusive Scan		
			Example using MPI_SCAN		
	5.12		ocking Collective Operations		
	0.11		Nonblocking Barrier Synchronization		
			Nonblocking Broadcast		
			Example using MPI_IBCAST		
		5.12.3	Nonblocking Gather		
			Nonblocking Scatter		
			Nonblocking Gather-to-all		
			Nonblocking All-to-All Scatter/Gather		

		5.12.7	Nonblocking Reduce	209
		5.12.8	Nonblocking All-Reduce	210
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks	211
		5.12.10	Nonblocking Reduce-Scatter	212
		5.12.11	Nonblocking Inclusive Scan	213
				214
	5.13		tness	214
6			, , , , , , , , , , , , , , , , , , ,	223
	6.1			223
		6.1.1		223
		6.1.2	11	224
	6.2		*	226
		6.2.1	1	226
		6.2.2	Contexts	226
		6.2.3		227
		6.2.4	Predefined Intra-Communicators	227
	6.3	Group	0	228
		6.3.1	Group Accessors	228
		6.3.2	*	230
		6.3.3	Group Destructors	235
	6.4	Comm	unicator Management	235
		6.4.1	Communicator Accessors	235
		6.4.2		237
		6.4.3		248
		6.4.4	Communicator Info	248
	6.5	Motiva	ating Examples	250
		6.5.1	Current Practice #1	250
		6.5.2	Current Practice $#2$	251
		6.5.3	(Approximate) Current Practice $#3$	251
		6.5.4		252
		6.5.5		253
		6.5.6	5 I 11	255
	6.6	Inter-C	Communication	257
		6.6.1	Inter-communicator Accessors	259
		6.6.2	Inter-communicator Operations	260
		6.6.3	Inter-Communication Examples	263
			Example 1: Three-Group "Pipeline"	263
			Example 2: Three-Group "Ring"	264
	6.7	Cachin	ng	265
		6.7.1	Functionality	266
		6.7.2	Communicators	267
		6.7.3	Windows	272
		6.7.4	Datatypes	275
		6.7.5	Error Class for Invalid Keyval	279
		6.7.6	Attributes Example	279
	6.8	Namin	g Objects	281
	6.9	Forma	lizing the Loosely Synchronous Model	285

		6.9.1	Basic Statements		285
		6.9.2	Models of Execution		286
			Static Communicator Allocation		286
			Dynamic Communicator Allocation		286
			The General Case		287
7	Pro	$\mathbf{cess} \ \mathbf{T}$	opologies		<b>289</b>
	7.1	Introd	luction		289
	7.2	Virtua	al Topologies		290
	7.3	Embe	dding in MPI		290
	7.4	Overv	riew of the Functions		290
	7.5	Topol	ogy Constructors		292
		7.5.1	Cartesian Constructor		292
		7.5.2	Cartesian Convenience Function: MPI_DIMS_CREATE		293
		7.5.3	Graph Constructor		294
		7.5.4	Distributed Graph Constructor		296
		7.5.5	Topology Inquiry Functions		302
		7.5.6	Cartesian Shift Coordinates		310
		7.5.7	Partitioning of Cartesian Structures		311
		7.5.8	Low-Level Topology Functions		312
	7.6		borhood Collective Communication		314
	1.0	7.6.1	Neighborhood Gather		315
		7.6.2	Neighbor Alltoall		318
	7.7		locking Neighborhood Communication		$310 \\ 323$
	1.1				
		7.7.1	Nonblocking Neighborhood Gather		324
	70	7.7.2	Nonblocking Neighborhood Alltoall		326
	7.8	An Ap	pplication Example	• •	329
8	MPI	Envi	ronmental Management		333
	8.1		mentation Information		333
	0.1	8.1.1	Version Inquiries		333
		8.1.2	Environmental Inquiries		334
		0.1.2	Tag Values		335
			Host Bank	•••	335
			IO Rank	• •	335
			Clock Synchronization		336
					336
	0.0	<b>М</b>	Inquire Processor Name		
	8.2		Pry Allocation		337
	8.3		Handling		340
		8.3.1	Error Handlers for Communicators		341
		8.3.2	Error Handlers for Windows		343
		8.3.3	Error Handlers for Files		345
		8.3.4	Freeing Errorhandlers and Retrieving Error Strings		346
	8.4		Codes and Classes		347
	8.5		Classes, Error Codes, and Error Handlers		350
	8.6	Timer	s and Synchronization		354
	8.7	Startu			355
		8.7.1	Allowing User Functions at Process Termination		360

			9	361
	8.8	Portab	ble MPI Process Startup	361
9	The	Info C	)bject ::	365
10	Pro	cess C	reation and Management	371
	10.1	Introd	$\operatorname{uction}$	371
	10.2	The D	ynamic Process Model	372
		10.2.1	Starting Processes	372
		10.2.2	The Runtime Environment	372
	10.3			374
		10.3.1	Processes in MPI	374
		10.3.2	Starting Processes and Establishing Communication	374
		10.3.3	Starting Multiple Executables and Establishing Communication	379
		10.3.4	Reserved Keys	382
		10.3.5	Spawn Example	383
			Manager-worker Example Using MPI_COMM_SPAWN	383
	10.4	Establ	ishing Communication	385
		10.4.1	Names, Addresses, Ports, and All That	385
		10.4.2	Server Routines	386
		10.4.3	Client Routines	388
		10.4.4	Name Publishing	390
		10.4.5	Reserved Key Values	392
		10.4.6	Client/Server Examples	392
			Simplest Example — Completely Portable.	392
			Ocean/Atmosphere — Relies on Name Publishing	393
				393
	10.5	Other	Functionality	395
		10.5.1	Universe Size	395
		10.5.2	Singleton MPI_INIT	396
		10.5.3	MPI_APPNUM	396
		10.5.4	Releasing Connections	397
		10.5.5	Another Way to Establish MPI Communication	399
11	One	-Sided	Communications	401
	11.1	Introd		401
	11.2	Initiali		402
				403
			5	405
		11.2.3	Window That Allocates Shared Memory	407
				410
				413
				413
				415
	11.3	Comm		416
		11.3.1		417
		11.3.2		420
		11.3.3	Examples for Communication Calls	421

	11.3.4 Accumulate Functions
	Accumulate Function
	Get Accumulate Function
	Fetch and Op Function
	Compare and Swap Function
	11.3.5 Request-based RMA Communication Operations
11.4	Memory Model
11.5	Synchronization Calls
	11.5.1 Fence
	11.5.2 General Active Target Synchronization
	11.5.3 Lock
	11.5.4 Flush and Sync
	11.5.5 Assertions
	11.5.6 Miscellaneous Clarifications
11.6	Error Handling
	11.6.1 Error Handlers
	11.6.2 Error Classes
11.7	Semantics and Correctness
	11.7.1 Atomicity
	11.7.2 Ordering
	11.7.3 Progress
	11.7.4 Registers and Compiler Optimizations
11.8	Examples
<b>12</b> Exte	ernal Interfaces 473
12.1	Introduction
12.2	Generalized Requests
	12.2.1 Examples
12.3	Associating Information with Status
12.4	MPI and Threads
	12.4.1 General
	12.4.2 Clarifications
	12.4.3 Initialization
13 I/O	489
13.1	Introduction
	13.1.1 Definitions
13.2	File Manipulation    491
	13.2.1 Opening a File
	13.2.2 Closing a File
	13.2.3 Deleting a File
	13.2.4 Resizing a File
	13.2.5 Preallocating Space for a File
	13.2.6 Querying the Size of a File 496
	13.2.7 Querying File Parameters
	13.2.8 File Info
	Reserved File Hints 500
13.3	File Views

13.4	Data 4	Access	 	 			•		. 504
	13.4.1	Data Access Routines	 	 					. 504
		Positioning	 	 					. 505
13.5			 	 					. 505
		Synchronism	 	 					. 500
		Coordination	 	 					. 500
		Data Access Conventions	 	 					. 500
	13.5.1	Data Access with Explicit Offsets	 	 					. 50'
		Data Access with Individual File Pointers .							
		Data Access with Shared File Pointers							
		Noncollective Operations							
		Collective Operations							
		Seek							
	1354	Split Collective Data Access Routines							
13.6		teroperability							
13.7		· · · · · · · · · · · · · · · · · · ·							
10.1		Datatypes for File Interoperability							
		External Data Representation: "external32"							
		User-Defined Data Representations							
	10.7.0	Extent Callback							
	1974	Datarep Conversion Functions							
19.0		Matching Data Representations							
13.8		stency and Semantics							
		File Consistency							
		Random Access vs. Sequential Files							
		Progress							
		Collective File Operations							
		Type Matching							
		Miscellaneous Clarifications							
		MPI_Offset Type							
		Logical vs. Physical File Layout							
		File Size							
	13.8.10	) Examples $\ldots$							
		Asynchronous I/O							
		rror Handling							
13.10	OI/O E	rror Classes	 	 	•	• •			. 551
13.1	1Exam <sub>l</sub>	ples	 	 					. 555
	13.11.	I Double Buffering with Split Collective I/O	 	 					. 552
	13.11.2	2 Subarray Filetype Constructor	 	 					. 555
13.12	2		 	 					. 555
	l Supp								55'
		uction							
14.2		ng Interface							
		Requirements							
		Discussion							
		Logic of the Design							
	14.2.4	Miscellaneous Control of Profiling	 	 	•		•	•	. 559

	14.2.5 Profiler Implementation Example	. 560
	14.2.6 MPI Library Implementation Example	. 560
	Systems with Weak Symbols	
	Systems Without Weak Symbols	. 561
	14.2.7 Complications	
	Multiple Counting	
	Linker Oddities	
	Fortran Support Methods	
	14.2.8 Multiple Levels of Interception	
14.3	The MPI Tool Information Interface	
11.0	14.3.1 Verbosity Levels	
	14.3.2 Binding MPI Tool Information Interface Variables to MPI Objects	
	14.3.3 Convention for Returning Strings	
	14.3.4 Initialization and Finalization	
	14.3.5 Datatype System $\ldots$	
	14.3.6 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	
	Performance Variables	
	Performance Variable Query Functions	
	Performance Experiment Sessions	
	Starting and Stopping of Performance Variables	
	Example: Tool to Detect Receives with Long Unexpected Messag	
	Queues	
	14.3.8 Variable Categorization	
	14.3.9 Return Codes for the MPI Tool Information Interface	
	14.3.10 Profiling Interface	. 590
15 Dop	recated Interfaces	594
	Deprecated since MPI-2.0	001
	Deprecated since MPI-2.2	
	Deprecated since MPI-4.0	
10.0		. 001
16 Ren	oved Interfaces	599
16.1	Removed MPI-1 Bindings	. 599
	16.1.1 Overview	
	16.1.2 Removed MPI-1 Functions	. 599
	16.1.3 Removed MPI-1 Datatypes	. 599
	16.1.4 Removed MPI-1 Constants	
	16.1.5 Removed MPI-1 Callback Prototypes	
16.2	C++ Bindings	

17 Language Bindings	601
17.1 Fortran Support	. 601
17.1.1 Overview	. 601
17.1.2 Fortran Support Through the mpi_f08 Module	. 603
17.1.3 Fortran Support Through the mpi Module	
17.1.4 Fortran Support Through the mpif.h Include File	
17.1.5 Interface Specifications, Linker Names and the Profiling Interface .	
17.1.6 MPI for Different Fortran Standard Versions	
17.1.7 Requirements on Fortran Compilers	. 617
17.1.8 Additional Support for Fortran Register-Memory-Synchronization	
17.1.9 Additional Support for Fortran Numeric Intrinsic Types	. 620
Parameterized Datatypes with Specified Precision and Exponent Ra	
Support for Size-specific MPI Datatypes	-
Communication With Size-specific Types	
17.1.10 Problems With Fortran Bindings for MPI	
17.1.11 Problems Due to Strong Typing	
17.1.12 Problems Due to Data Copying and Sequence Association with Sub	
script Triplets	
17.1.13 Problems Due to Data Copying and Sequence Association with Vector	
Subscripts	
17.1.14 Special Constants	
17.1.15 Fortran Derived Types	
17.1.16 Optimization Problems, an Overview	
17.1.17 Problems with Code Movement and Register Optimization	
Nonblocking Operations	
One-sided Communication	
MPI_BOTTOM and Combining Independent Variables in Datatype	
Solutions	
The Fortran ASYNCHRONOUS Attribute	
Calling MPI_F_SYNC_REG	
A User Defined Routine Instead of MPI_F_SYNC_REG	
Module Variables and COMMON Blocks	
The (Poorly Performing) Fortran VOLATILE Attribute	
The Fortran TARGET Attribute	
17.1.18 Temporary Data Movement and Temporary Memory Modification	
17.1.19 Permanent Data Movement $\dots$	
17.1.20 Comparison with C	
17.2 Language Interoperability	
17.2.1 Introduction	
17.2.2 Assumptions $\dots \dots \dots$	
17.2.3 Initialization	
17.2.4 Transfer of Handles	
17.2.5 Status	
17.2.6 MPI Opaque Objects	
Datatypes	
Callback Functions	
Error Handlers	
Reduce Operations	. 657

		17.2.7	Attributes	657	
		17.2.8	Extra-State	661	
		17.2.9	Constants	661	
		17.2.10	)Interlanguage Communication	662	
Α	Lan	guage	Bindings Summary	665	
			d Values and Handles	665	
		A.1.1	Defined Constants		
		A.1.2	Types	679	
		A.1.3	Prototype Definitions	680	
			C Bindings	680	
			Fortran 2008 Bindings with the mpi_f08 Module	680	
			Fortran Bindings with the deprecated mpif.h or the mpi Module	683	$\operatorname{tick}$
		A.1.4	Deprecated Prototype Definitions	685	
		A.1.5	Info Keys	686	
		A.1.6	Info Values		
	A.2	C Bine	lings	687	
		A.2.1	Point-to-Point Communication C Bindings		
		A.2.2	Datatypes C Bindings		
		A.2.3	Collective Communication C Bindings		
		A.2.4	Groups, Contexts, Communicators, and Caching C Bindings		
		A.2.5	Process Topologies C Bindings		
		A.2.6	MPI Environmental Management C Bindings		
		A.2.7	The Info Object C Bindings		
		A.2.8	Process Creation and Management C Bindings		
		A.2.9	One-Sided Communications C Bindings		
			External Interfaces C Bindings		
			I/O C Bindings		
			Language Bindings C Bindings		
			Tools / Profiling Interface C Bindings		
			Tools / MPI Tool Information Interface C Bindings		
	12		n 2008 Bindings with the mpi_f08 Module		
	А.Э		Point-to-Point Communication Fortran 2008 Bindings		
		A.3.2	Datatypes Fortran 2008 Bindings		
		A.3.3	Collective Communication Fortran 2008 Bindings		
		A.3.4	Groups, Contexts, Communicators, and Caching Fortran 2008 Bindin		
		A.3.5	Process Topologies Fortran 2008 Bindings	<u> </u>	
		A.3.6	MPI Environmental Management Fortran 2008 Bindings		
		A.3.7	The Info Object Fortran 2008 Bindings		
		A.3.8	Process Creation and Management Fortran 2008 Bindings		
		A.3.9	One-Sided Communications Fortran 2008 Bindings		
			External Interfaces Fortran 2008 Bindings		
			I/O Fortran 2008 Bindings		
			Language Bindings Fortran 2008 Bindings		
			Tools / Profiling Interface Fortran 2008 Bindings		
	A.4		n Bindings with mpif.h or the mpi Module		
			Point-to-Point Communication Fortran Bindings		

ticketWG.

	A.4.2	Datatypes Fortran Bindings	761	
	A.4.3	Collective Communication Fortran Bindings	763	
	A.4.4	Groups, Contexts, Communicators, and Caching Fortran Bindings .	767	
	A.4.5	Process Topologies Fortran Bindings	771	
	A.4.6	MPI Environmental Management Fortran Bindings	774	
	A.4.7	The Info Object Fortran Bindings	776	
	A.4.8	Process Creation and Management Fortran Bindings	777	
	A.4.9	One-Sided Communications Fortran Bindings	778	
	A.4.10	External Interfaces Fortran Bindings	781	
	A.4.11	I/O Fortran Bindings	782	
	A.4.12	Language Bindings Fortran Bindings	786	
	A.4.13	Tools / Profiling Interface Fortran Bindings	787	
	A.4.14	Deprecated Fortran Bindings	787	
B Cha	ange-Lo	)g	789	
B.1	0	es from Version 2.2 to Version 3.0	789	
		Fixes to Errata in Previous Versions of MPI	789	
	B.1.2	Changes in MPI-3.0	790	
B.2	Chang	es from Version 2.1 to Version 2.2	795	
B.3	-	es from Version 2.0 to Version 2.1	797	
Bibliog	graphy		803	
Examp	oles Inc	lex	808	
MPI C	Constan	at and Predefined Handle Index	811	
MPI I	MPI Declarations Index			
MPI C	MPI Callback Function Prototype Index			
MPI F	MPI Function Index			

# List of Figures

5.1	Collective comminucations, an overview	143
5.2	Intercommunicator allgather	146
5.3	Intercommunicator reduce-scatter	147
5.4	Gather example	153
5.5	Gatherv example with strides	154
5.6	Gatherv example, 2-dimensional	155
5.7	Gatherv example, 2-dimensional, subarrays with different sizes	156
5.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	158
5.9	Scatter example	163
5.10	Scattery example with strides	163
5.11	Scattery example with different strides and counts	164
5.12	Race conditions with point-to-point and collective communications	217
5.13	Overlapping Communicators Example	221
6.1	Intercommunicator creation using MPI_COMM_CREATE	242
6.2	Intercommunicator construction with MPI_COMM_SPLIT	246
6.3	Three-group pipeline	263
6.4	Three-group ring	264
<ul><li>7.1</li><li>7.2</li><li>7.3</li><li>7.4</li></ul>	FIXME: You cannot use the label command without a captionSet-up of process structure for two-dimensional parallel Poisson solver.Communication routine with local data copying and sparse neighborhoodall-to-all.Communication routine with sparse neighborhood all-to-all-w and withoutlocal data copying.	316 330 331 332
11.1	Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows	435
11.2	Active target communication	438
11.3	Active target communication, with weak synchronization	439
11.4	Passive target communication	440
11.5	Active target communication with several processes	443
	Symmetric communication	462
11.7	Deadlock situation	462
	No deadlock	462
		100
	Etypes and filetypes	490
	Partitioning a file among parallel processes	490
13.3	Displacements	503

13.4	Example array file layout	554
13.5	Example local array filetype for process 1	554
17.1	Status conversion routines	653

# List of Tables

2.1	Deprecated and Removed constructs	18
3.1	Predefined MPI datatypes corresponding to Fortran datatypes	2!
3.2	Predefined MPI datatypes corresponding to C datatypes $\ldots \ldots \ldots$	20
3.3	Predefined $MPI$ data types corresponding to both C and Fortran data types .	2'
3.4	Predefined MPI datatypes corresponding to C++ datatypes $\ldots \ldots \ldots$	2'
4.1	combiner values returned from MPI_TYPE_GET_ENVELOPE $\ldots \ldots \ldots$	11
6.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	25
8.1	Error classes (Part 1)	343
8.2	Error classes (Part 2)	349
11.1	C types of attribute value argument to $MPI_WIN_GET_ATTR$ and	
	MPI_WIN_SET_ATTR	41
11.2	Error classes in one-sided communication routines	452
13.1	Data access routines	50
	"external32" sizes of predefined datatypes	53
13.3	I/O Error Classes	55
	MPI tool information interface verbosity levels	56
	Constants to identify associations of variables	56
	MPI datatypes that can be used by the MPI tool information interface	56
	Scopes for control variables	57
14.5	Return codes used in functions of the $MPI$ tool information interface $\ldots$	59
16.1	Removed MPI-1 functions and their replacements	59
	Removed MPI-1 datatypes and their replacements	60
	Removed MPI-1 constants	60
16.4	Removed MPI-1 callback prototypes and their replacements	60
17.1	Occurrence of Fortran optimization problems	63

 $45 \\ 46$ 

# 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
- 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  $\mathsf{MPI-1.0}$  and  $\mathsf{MPI-1.1}$  process not mentioned above.

xx

 $\overline{7}$ 

 $^{24}$ 

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz	1
Scott Berrym	an Rob Bjornson	Nathan Doss	Anne Elster	2
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower	3
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison	4
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson	5
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga	6
James Kohl Peter Madam	Susan Krauss	Bob Leary	Arthur Maccabe	7 8
Charles Mosh	0	Oliver McBryan Peter Pacheco	Phil McKinley Howard Palmer	9
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison	10
Erich Schikut	0 0	Alan Sussman	Robert Tomlinson	11
Robert G. Vo		Stephen Wheat	Steve Zenith	12
			Laboratory made the draft avail-	13
			al in distributing the document.	14
			t by ARPA and NSF under grant	15
			Technology Center Cooperative	16
0	CCR-8809615, and by 1 P6643 (PPPE).	the Commission of th	e European Community through	17 18
Espin project	$\Gamma 0043 (\Gamma \Gamma \Gamma E).$			18
MPI-1.2 and N	/IPI_2 ∩·			20
				21
Those who ser	ved as primary coordina	ators in $MPI-1.2$ and	MPI-2.0 are:	22
• Ewing L	usk, Convener and Meet	ing Chair		23
0				24
• Steve Hu	ss-Lederman, Editor			25
• Ewing L	ısk, Miscellany			26 27
• Bill Saphir, Process Creation and Management				29
• Marc Snir, One-Sided Communications				30
				31
• Bill Grou	p and Anthony Skjellur	m, Extended Collect	ive Operations	32
• Steve Hu	ss-Lederman, External	Interfaces		33
	I/O			34 35
• Bill Nitz	perg, $1/O$			36
• Andrew 1	Lumsdaine, Bill Saphir,	and Jeff Squyres, L	anguage Bindings	37
• Anthony	Skjellum and Arkady K	Zanevsky Real-Time		38
• minimizing	Shjohum and minady is	tane vory, recar rime		39
The follow	ing list includes some o	f the active particip	ants who attended MPI-2 Forum	40
meetings and a	are not mentioned above	9.		41
				42 43
				44
				45
				46

1	Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar
2	Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz
3	Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen
4	Ying Chen	Albert Cheng	Yong Cho	Joel Clark
5	Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie
6	Zhenqian Cui	Suresh Damodaran-Kar		Raja Daoud
7	Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra
8	Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon
9	Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman
10	Ian Foster	Hubertus Franke	Richard Frost	Al Geist
11	Robert George	David Greenberg	John Hagedorn	Kei Harada
12	Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson
13	Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones
14	Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica
15	Steve Landherr	Mario Lauria	Mark Law	Juan Leon
16	Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams
17	John May	Oliver McBryan	Brian McCandless	Tyce McLarty
18	Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha
19	Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco
20	Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce
21	Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov
22	James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee
23	Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha
24	Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman
25	Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale
26	David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur
27	Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon
28	David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong
29	Dave Wright			
30		also acknowledges and ap	preciates the valuable	e input from people via
31	e-mail and in person.			
32		titutions supported the $\mathbb{N}$	MPI-2 effort through	time and travel support
33	for the people listed a			
34	Argonne Nationa	0		
35	Bolt, Beranek, an			
36	California Institu	01		
37	Center for Comp	-		
38	Convex Compute	er Corporation		
39	Cray Research			
40	Digital Equipmen	_		
41	Dolphin Interconnect Solutions, Inc.			
42	-	lel Computing Centre		
43	General Electric			
44		l Research Center for Inf	ormation lechnology	
45	Hewlett-Packard			
46	Hitachi			
47	Hughes Aircraft			
48	Intel Corporation	1		

International Business Machines	1
Khoral Research	2
Lawrence Livermore National Laboratory	3
Los Alamos National Laboratory	4
MPI Software Techology, Inc.	5
Mississippi State University	6
NEC Corporation	7
National Aeronautics and Space Administration	8
National Energy Research Scientific Computing Center	9
National Institute of Standards and Technology	10
National Oceanic and Atmospheric Adminstration	11
Oak Ridge National Laboratory	12
Ohio State University	13
PALLAS GmbH	14
Pacific Northwest National Laboratory	15
Pratt & Whitney	16
San Diego Supercomputer Center	17
Sanders, A Lockheed-Martin Company	18
Sandia National Laboratories	19
Schlumberger	20
Scientific Computing Associates, Inc.	21
Silicon Graphics Incorporated	22
Sky Computers	23
Sun Microsystems Computer Corporation	24
Syracuse University	25
The MITRE Corporation	26
Thinking Machines Corporation	27
United States Navy	28
University of Colorado	29
University of Denver	30
University of Houston	31
University of Illinois	32
University of Maryland	33
University of Notre Dame	34
University of San Fransisco	35
University of Stuttgart Computing Center	36
University of Wisconsin	37
MPI-2 operated on a very tight budget (in reality, it had no budget when the first	38

MPI-2 operated on a very tight budget (in reality, it had no budget when the first 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 DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and Esprit under project HPC Standards (21111) for European participants.

 $^{41}$ 

MPI-1.3 and MPI-2.1:	44
	45
The editors and organizers of the combined documents have been:	46
• Richard Graham, Convener and Meeting Chair	47
• Identify Granam, Convence and Meeting Chair	48

1	• Jack Dongarra, Steering Committee
2 3	• Al Geist, Steering Committee
4 5	• Bill Gropp, Steering Committee
6	• Rainer Keller, Merge of MPI-1.3
7 8	• Andrew Lumsdaine, Steering Committee
9 10 11	• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)
12 13	<ul> <li>Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)</li> </ul>
14 15 16	All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:
17 18	• Bill Gropp, Front matter, Introduction, and Bibliography
19	• Richard Graham, Point-to-Point Communication
20 21	• Adam Moody, Collective Communication
22 23	• Richard Treumann, Groups, Contexts, and Communicators
26 24 25	• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions
26 27	• George Bosilca, Environmental Management
28 29	• David Solt, Process Creation and Management
30	• Bronis R. de Supinski, External Interfaces, and Profiling
31 32	• Rajeev Thakur, I/O
33	• Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary
35	• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log
36 37	• Alexander Supalov and Denis Nagorny, Annex Language Bindings
38 39 40 41	The following list includes some of the active participants who attended MPI-2 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.
42	
43 44	
44 45	
46	
47	
48	

Purushotham V. Bangalore	Brian Barrett	1
Christian Bell	Robert Blackmore	2
Ron Brightwell	Jeffrey Brown	3
Jonathan Carter	Nathan DeBardeleben	4
Gabor Dozsa	Edric Ellis	5
Edgar Gabriel	Patrick Geoffray	6
Dave Goodell	Erez Haba	7
Thomas Herault	Steve Hodson	8
Joshua Hursey	Yann Kalemkarian	9
Quincey Koziol	Sameer Kumar	10
Kannan Narasimhan	Mark Pagel	11
Steve Poole	Howard Pritchard	12
Hubert Ritzdorf	Rob Ross	13
Brian Smith	Vinod Tipparaju	14
Keith Underwood		15
so acknowledges and apprecia	tes the valuable input from people via	16
		17
tutions supported the MPI-2 of	effort through time and travel support	18
oove.		19
Laboratory		20
		21
		22
		23
		24
		25
Research		26
		27
le Recherche en Informatique	et Automatique (INRIA)	28
		29
National Laboratory		30
re National Laboratory		31
nal Laboratory		32
		33
ogies		34
		35
		36
Europe, NEC Europe Ltd.		37
al Laboratory		38
sity		39
National Laboratory		40
on		41
aboratories		42
		43
ncorporated		44
, Inc.		45
ama at Birmingham		46
ston		47
ois at Urbana-Champaign		48
	Christian Bell Ron Brightwell Jonathan Carter Gabor Dozsa Edgar Gabriel Dave Goodell Thomas Herault Joshua Hursey Quincey Koziol Kannan Narasimhan Steve Poole Hubert Ritzdorf Brian Smith Keith Underwood so acknowledges and apprecia tutions supported the MPI-2 of ove. Laboratory :. Research le Recherche en Informatique 'National Laboratory re National Laboratory nal Laboratory gies Europe, NEC Europe Ltd. al Laboratory sity National Laboratory maboratories ncorporated Inc. ama at Birmingham ton	Christian Bell Robert Blackmore Ron Brightwell Jeffrey Brown Jonathan Carter Nathan DeBardeleben Gabor Dozsa Edric Ellis Edgar Gabriel Patrick Geoffray Dave Goodell Erez Haba Thomas Herault Steve Hodson Joshua Hursey Yann Kalemkarian Quincey Koziol Sameer Kumar Kannan Narasimhan Mark Pagel Steve Poole Howard Pritchard Hubert Ritzdorf Rob Ross Brian Smith Vinod Tipparaju Keith Underwood so acknowledges and appreciates the valuable input from people via tutions supported the MPI-2 effort through time and travel support ove. Laboratory Research Research Research Research Research Research Laboratory giges Europe, NEC Europe Ltd. al Laboratory sity National Laboratory maboratories corporated Inc. ama at Birmingham ton

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

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

1	Ohio State University		
2	Pacific Northwest National Laboratory		
3	QLogic Corporation		
4	RunTime Computing Solutions, LLC		
5	Sandia National Laboratories		
6	SiCortex, Inc.		
7	Silicon Graphics Inc.		
8	Sun Microsystems, Inc.		
9	Tokyo Institute of Technology		
10	University of Alabama at Birmingham		
11	University of Houston		
12	University of Illinois at Urbana-Champaign		
13	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)		
14	University of Tennessee, Knoxville		
15	University of Tokyo		
16	University of Wisconsin		
17	Funding for the MPI Forum meetings was partially supported by awards $\#$ CCF-0816909		
18	and $\#$ CCF-1144042 from the National Science Foundation. In addition, the HDF Group		
19	provided travel support for one U.S. academic.		
20			
21	MPI-3:		
22	MPI-3 is a significant effort to extend and modernize the MPI Standard.		
23	The editors and organizers of the MPI-3 have been:		
24	The editors and organizers of the Wir PS have been.		
25	• William Gropp, Steering committee, Front matter, Introduction, Groups, Contexts,		
26	and Communicators, One-Sided Communications, and Bibliography		
27			
28	• Richard Graham, Steering committee, Point-to-Point Communication, Meeting Con-		
29 30	vener, and MPI-3 chair		
31	• Torsten Hoefler, Collective Communication, One-Sided Communications, and Process		
32	Topologies		
33			
34	• George Bosilca, Datatypes and Environmental Management		
35	• David Solt, Process Creation and Management		
36	• David Soft, 1 locess creation and Management		
37	• Bronis R. de Supinski, External Interfaces and Tool Support		
38			
39	• Rajeev Thakur, I/O and One-Sided Communications		
40	• Darius Buntinas, Info Object		
41			
42	• Jeffrey M. Squyres, Language Bindings and MPI-3 Secretary		
43	• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings,		
44	• Kon Rabensenner, Steering committee, Terms and Demitions, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings		
45	Depression Functions, Annex Change-Log, and Annex Danguage Dinulligs		
46	• Craig Rasmussen, Fortran Bindings		
47			
48			

The following list includes so	me of the active partici	pants who attended MPI-3 Forum	1		
meetings or participated in the e-	=	-	2		
Tatsuya Abe	Tomoya Adachi	Sadaf Alam	3		
Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore	4		
Brian Barrett	<b>Richard Barrett</b>	Robert Blackmore	5		
Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky	6		
Jed Brown	Darius Buntinas	Devendar Bureddy	7		
Arno Candel	George Carr	Mohamad Chaarawi	8		
Raghunath Raja Chandrasekar	James Dinan	Terry Dontje	9		
Edgar Gabriel	Balazs Gerofi	Brice Goglin	10		
David Goodell	Manjunath Gorentla	Erez Haba	11		
Jeff Hammond	Thomas Herault	Marc-André Hermanns	12		
Jennifer Herrett-Skjellum	Nathan Hjelm	Atsushi Hori	13		
Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa	14		
Nysal Jan	Bin Jia	Hideyuki Jitsumoto	15		
Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima	16		
Chulho Kim	Dries Kimpe	Christof Klausecker	17		
Alice Koniges	Quincey Koziol	Dieter Kranzlmueller	18		
Manojkumar Krishnan	Sameer Kumar	Eric Lantz	19		
Jay Lofstead	Bill Long	Andrew Lumsdaine	20		
Miao Luo	Ewing Lusk	Adam Moody	21		
Nick M. Maclaren	Amith Mamidala	Guillaume Mercier	22		
Scott McMillan	Douglas Miller	Kathryn Mohror	23		
Tim Murray	Tomotake Nakamura	Takeshi Nanri	24		
Steve Oyanagi	Mark Pagel	Swann Perarnau	25		
Sreeram Potluri	Howard Pritchard	Rolf Riesen	26		
Hubert Ritzdorf	Kuninobu Sasaki	Timo Schneider	27		
Martin Schulz	Gilad Shainer	Christian Siebert	28		
Anthony Skjellum	Brian Smith	Marc Snir	29		
Raffaele Giuseppe Solca	Shinji Sumimoto	Alexander Supalov	30		
Sayantan Sur	Masamichi Takagi	Fabian Tillier	31		
Vinod Tipparaju	Jesper Larsson Träff	Richard Treumann	32		
Keith Underwood	Rolf Vandevaart	Anh Vo	33		
Abhinav Vishnu	Min Xie	Enqiang Zhou	34		
The MPI Forum also acknowl	edges and appreciates t	the valuable input from people via	35		
e-mail and in person.			36		
	_	dback during the public comment	37		
period. In particular, the Forum would like to thank Jeremiah Wilcock for providing detailed					
comments on the entire draft standard.					
The following institutions supported the $MPI-3$ effort through time and travel support					
for the people listed above.					
Argonne National Laboratory					
Bull					
Cisco Systems, Inc.					
Cray Inc.					
CSCS			46		
ETH Zurich			47		

Fujitsu Ltd.

- <sup>1</sup> German Research School for Simulation Sciences
- <sup>2</sup> The HDF Group
- <sup>3</sup> Hewlett-Packard
- <sup>4</sup> International Business Machines
- <sup>5</sup> IBM India Private Ltd
- <sup>6</sup> Indiana University
- <sup>7</sup> Institut National de Recherche en Informatique et Automatique (INRIA)
- <sup>8</sup> Institute for Advanced Science & Engineering Corporation
- <sup>9</sup> Intel Corporation
- <sup>10</sup> Lawrence Berkeley National Laboratory
- <sup>11</sup> Lawrence Livermore National Laboratory
- <sup>12</sup> Los Alamos National Laboratory
- <sup>13</sup> Mellanox Technologies, Inc.
- <sup>14</sup> Microsoft Corporation
- <sup>15</sup> NEC Corporation
- <sup>16</sup> National Oceanic and Atmospheric Administration, Global Systems Division
- <sup>17</sup> NVIDIA Corporation
- <sup>18</sup> Oak Ridge National Laboratory
- <sup>19</sup> The Ohio State University
- <sup>20</sup> Oracle America
- <sup>21</sup> Platform Computing
- <sup>22</sup> RIKEN AICS
- <sup>23</sup> RunTime Computing Solutions, LLC
- <sup>24</sup> Sandia National Laboratories
- <sup>25</sup> Technical University of Chemnitz
- <sup>26</sup> Tokyo Institute of Technology
- <sup>27</sup> University of Alabama at Birmingham
- <sup>28</sup> University of Chicago
- <sup>29</sup> University of Houston
- <sup>30</sup> University of Illinois at Urbana-Champaign
- <sup>31</sup> University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
- <sup>32</sup> University of Tennessee, Knoxville
- <sup>33</sup> University of Tokyo
- $^{34}$  Funding for the MPI Forum meetings was partially supported by awards #CCF-0816909
- $^{35}$   $\,$  and  $\# \rm CCF-1144042$  from the National Science Foundation. In addition, the HDF Group
- <sup>36</sup> and Sandia National Laboratories provided travel support for one U.S. academic each.
- 37
- 38
- 39 40
- 40
- 42

- 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 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.

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 

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 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-processors, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C 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

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

18 The MPI standardization effort involved about 60 people from 40 organizations mainly 19from the United States and Europe. Most of the major vendors of concurrent computers 20were involved in MPI, along with researchers from universities, government laboratories, and 21industry. The standardization process began with the Workshop on Standards for Message-22Passing in a Distributed Memory Environment, sponsored by the Center for Research on 23Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia 60. At this workshop  $^{24}$ the basic features essential to a standard message-passing interface were discussed, and a 25working group established to continue the standardization process. 26

A preliminary draft proposal, known as MPI-1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [18]. MPI-1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI-1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI-1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at 34which it was decided to place the standardization process on a more formal footing, and to 35 generally adopt the procedures and organization of the High Performance Fortran Forum. 36 Subcommittees were formed for the major component areas of the standard, and an email 37 discussion service established for each. In addition, the goal of producing a draft MPI 38 standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 39 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI 40 standard at the Supercomputing 93 conference in November 1993. These meetings and the 41 email discussion together constituted the MPI Forum, membership of which has been open 42to all members of the high performance computing community. 43

44

 $45 \\ 46$ 

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

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [22]. The first product of these deliberations

1

2

3

4

5 6

> 7 8

9 10

11 12

13

14

15

16

was Version 1.1 of the MPI specification, released in June of 1995 [23] (see http://www.mpi-forum.org for official MPI document releases). At that time, effort focused in five areas.

- 1. Further corrections and clarifications for the MPI-1.1 document.
- 2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.).
- 3. Completely new types of functionality (dynamic processes, one-sided communication, parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality."
- 4. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 to handle Fortran 90 issues.
- 5. Discussions of areas in which the MPI process and framework seem likely to be useful, but where more discussion and experience are needed before standardization (e.g., zero-copy semantics on shared-memory machines, real-time specifications).

Corrections and clarifications (items of type 1 in the above list) were collected in Chapter 3 of the MPI-2 document: "Version 1.2 of MPI." That chapter also contains the function for identifying the version number. Additions to MPI-1.1 (items of types 2, 3, and 4 in the above list) are in the remaining chapters of the MPI-2 document, and constitute the specification for MPI-2. Items of type 5 in the above list have been moved to a separate document, the "MPI Journal of Development" (JOD), and are not part of the MPI-2 Standard.

This structure makes it easy for users and implementors to understand what level of MPI compliance a given implementation has:

- MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of compliance. It means that the implementation conforms to the clarifications of MPI-1.1 function behavior given in Chapter 3 of the MPI-2 document. Some implementations may require changes to be MPI-1 compliant.
- MPI-2 compliance will mean compliance with all of MPI-2.1.
- The MPI Journal of Development is not part of the MPI Standard.

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.

#### Background of MPI-1.3 and MPI-2.1 1.4

42After 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 44MPI-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.

1 2

3

4

56

7

8 9

10 11

12

13

1415

16

17

18

19

2021

22

23

 $^{24}$ 25

26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

41

43

45

46

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

 $15 \\ 16$ 

17 18

19

20

21 22

23

24 25

26

27 28

29

30 31

32 33

34

35

36

37

## 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. The updates include the extension of collective operations to include nonblocking versions, extensions to the one-sided operations, and a new Fortran 2008 binding. In addition, the deprecated C++ bindings have been removed, as well as many of the deprecated routines and MPI objects (such as the MPI\_UB datatype).

38 39 40

41

## 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 and C (and access the C bindings from 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 management and inquiry,
- The Info object,
- Process creation and management,
- One-sided communication,
- External interfaces,
- Parallel file I/O,
- Language bindings for Fortran and C,
- Tool support.

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

## 1.10 What Is Not Included In The Standard?

- The standard does not specify:
  - Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
  - Program construction tools,
  - Debugging facilities.

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 selfimposed 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.

## 1.11 Organization of this Document

The following is a list of the remaining chapters in this document, along with a brief description of each.

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point-to-Point Communication, defines the basic, pairwise communication subset of MPI. *Send* and *receive* are found here, along with many associated functions designed to make basic communication powerful and efficient.
- 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.
- Chapter 5, Collective Communication, 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.
- 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 *communicator*.
- 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.

 $^{24}$ 

- Chapter 9, The Info Object, defines an opaque object, that is used as input 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, Tool Support, covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section 14.2 (Profiling Interface), which was a chapter in previous versions of MPI.
- Chapter 15, Deprecated Interfaces, 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, Removed Interfaces, describes routines and constructs that have been removed from MPI. These were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard.
- Chapter 17, Language Bindings, discusses Fortran issues, and describes language interoperability aspects between C and Fortran.

The Appendices are:

- Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for all MPI functions, constants, and types.
- Annex B, Change-Log, summarizes some changes since the previous version of the standard.
- Several Index pages show the locations of examples, constants and predefined handles, callback routine 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 MPI\_PACK\_EXTERNAL and MPI\_UNPACK\_EXTERNAL. The definition of an actual binding 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 during the MPI-2 development 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

 $^{24}$ 

1 2 3 4	• Chapter 2, Spawning Independent Processes, includes some elements of dynamic process 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.
5 6 7	• Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
8 9 10	• Chapter 4, Communicator ID, describes an approach to providing identifiers for communicators.
11 12 13	• Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
14 15 16	• Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.
17 18 19	• Chapter 7, Split Collective Communication, describes a specification for certain non- blocking collective operations.
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 33 34 35 36 37 38 39	• Chapter 8, Real-Time MPI, discusses MPI support for real time processing.
41 42 43	
44 45	
46	
47 48	

# 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.

 $45 \\ 46$ 

# 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.

- 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 MPI\_CLASS\_ACTION.
- 2. 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.

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.

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. MPI identifiers are limited to 30 characters (31 with the profiling interface). This is done to avoid exceeding the limit on some compilation systems.

# 2.3 Procedure Specification

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:

- 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,
- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

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 *references* is updated.

27 28

29

30 31 Rationale. 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. (*End of rationale.*)

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 processes and OUT by other processes. Such an argument is, syntactically, an INOUT argument 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.

<sup>44</sup> Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased
 <sup>45</sup> with any other argument passed to an MPI procedure. An example of argument aliasing in
 <sup>46</sup> C appears below. If we define a C procedure like this,

- 47
- 48

1

2

3

4

5

6

7

8

9 10 11

12 13

14

15 16

17 18

19 20

<pre>void copyIntBuffer( int *pin, int *pout, int len )</pre>	1
{    int i;	2
<pre>for (i=0; i<len; *pout++="*pin++;&lt;/pre" ++i)=""></len;></pre>	3
}	4
	5
then a call to it in the following code fragment has aliased arguments.	6
int a[10];	7
copyIntBuffer( a, a+3, 7);	8
15	9
Although the C language allows this, such usage of MPI procedures is forbidden unless	10
otherwise specified. Note that Fortran prohibits aliasing of arguments.	11
All MPI functions are first specified in the language-independent notation. Immediately	12
below this, language dependent bindings follow:	13
• The ISO C version of the function.	14
• The ISO C version of the function.	15
• The Fortran version used with USE mpi_f08.	16
	17
• The Fortran version of the same function used with USE mpi or the deprecated	<sup>18</sup> ticketWG.
INCLUDE 'mpif.h'.	19
"Fortran" in this document refers to Fortran 90 and higher; see Section 2.6.	20
Fortrair in this document refers to Fortrair 90 and higher, see Section 2.0.	21
	22
2.4 Semantic Terms	23
	24
When discussing MPI procedures the following semantic terms are used.	25
	26
<b>nonblocking</b> A procedure is nonblocking if the procedure may return before the operation	27
completes, and before the user is allowed to reuse resources (such as buffers) specified	28
in the call. A nonblocking request is <i>started</i> by the call that initiates it, e.g.,	29
	30
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,	31
communications. An <i>operation completes</i> when the user is allowed to reuse resources, and any output buffers have been updated; i.e., a call to $MPI\_TEST$ will return $flag =$	31 32
communications. An <i>operation completes</i> 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 <i>request is completed</i> by a call to wait, which returns, or a test or get status call	32 33
communications. An <i>operation completes</i> 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 <i>request is completed</i> 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	32 33 34
communications. An <i>operation completes</i> 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 <i>request is completed</i> 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	32 33 34 35
communications. An <i>operation completes</i> 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 <i>request is completed</i> 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 <i>freed</i> , and becomes <i>inactive</i> if it was persistent. A <i>communication completes</i> when all	32 33 34 35 36
communications. An <i>operation completes</i> 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 <i>request is completed</i> 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	32 33 34 35 36 37
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 <i>freed</i> , and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.	32 33 34 35 36 37 38
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed</li> </ul>	32 33 34 35 36 37 38 39
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 <i>freed</i> , and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.	32 33 34 35 36 37 38 39 40
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed</li> </ul>	32 33 34 35 36 37 38 39 40 41
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.</li> </ul>	32 33 34 35 36 37 38 39 40 41 42
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.</li> <li>local A procedure is local if completion of the procedure depends only on the local executing process.</li> </ul>	32 33 34 35 36 37 38 39 40 41 42 43
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.</li> <li>local A procedure is local if completion of the procedure depends only on the local executing process.</li> <li>non-local A procedure is non-local if completion of the operation may require the exe-</li> </ul>	32 33 34 35 36 37 38 39 40 41 42 43 44
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.</li> <li>local A procedure is local if completion of the procedure depends only on the local executing process.</li> <li>non-local A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require</li> </ul>	32 33 34 35 36 37 38 39 40 41 42 43 44
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.</li> <li>local A procedure is local if completion of the procedure depends only on the local executing process.</li> <li>non-local A procedure is non-local if completion of the operation may require the exe-</li> </ul>	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
<ul> <li>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 <i>freed</i>, and becomes <i>inactive</i> if it was persistent. A communication completes when all participating operations complete.</li> <li>blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.</li> <li>local A procedure is local if completion of the procedure depends only on the local executing process.</li> <li>non-local A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require</li> </ul>	32 33 34 35 36 37 38 39 40 41 42 43 44

	12 CHAPTER 2. MPI TERMS AND CONVENTIONS
1 2 3 4 5	<i>collective</i> 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.
6 7 8 9 10 11	<pre>predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT_INT, or MPI_PACKED) 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>
12	<i>derived</i> A derived datatype is any datatype that is not predefined.
<ol> <li>13</li> <li>14</li> <li>15</li> <li>16</li> <li>17</li> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> <li>27</li> <li>28</li> <li>29</li> <li>30</li> </ol>	<ul> <li>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,</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY,</li> <li>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,</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_HVECTOR or 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 architecture.</li> </ul>
30 31 32 33	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.

Data Types 2.5

#### 2.5.1 **Opaque Objects**

MPI manages system memory that is used for buffering messages and for storing internal 38 representations of various MPI objects such as groups, communicators, datatypes, etc. This 39 memory is not directly accessible to the user, and objects stored there are *opaque*: their size 40 and shape is not visible to the user. Opaque objects are accessed via *handles*, which exist 41in user space. MPI procedures that operate on opaque objects are passed handle arguments 42to access these objects. In addition to their use by MPI calls for object access, handles can 43 participate in assignments and comparisons. 44

ticketWG.  $_{45}$ 

In Fortran with USE mpi or the deprecated INCLUDE 'mpif.h', all handles have type INTEGER. In Fortran with USE mpi\_f08, and in 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

34

35 36

37

46

47

ticketWG. value is identical to the Fortran INTEGER value used in the mpi module and the deprecated mpif.h. The operators .EQ., .NE., == and /= are overloaded to allow the comparison of these handles. 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

The 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. (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]the mpi module or the deprecated mpif.h include file 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 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.

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

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects.

*Rationale.* This design hides the internal representation used for MPI data structures, thus allowing similar calls in C and Fortran. It also avoids conflicts with the typing

 $\mathbf{2}$ 

 $^{24}$ 

 $^{25}_{26}$  ticketWG.

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 handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated object is deallocated). It is the user's responsibility to avoid adding or deleting references to opaque objects, except as a result of MPI calls that allocate or deallocate such objects. (*End of advice to users.*)

Advice to implementors. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather then copies of its components; a call to MPI\_COMM\_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (*End of advice to implementors.*)

## 2.5.2 Array Arguments

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

  $^{31}$ 

## 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 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 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 switch and Fortran case/select statements) are:

٠		
	MPI_MAX_PROCESSOR_NAME	30
	MPI_MAX_LIBRARY_VERSION_STRING	31
	MPI_MAX_ERROR_STRING	32
	MPI_MAX_DATAREP_STRING	33
	MPI_MAX_INFO_KEY	34
	MPI_MAX_INFO_VAL	35
	MPI_MAX_OBJECT_NAME	36
	MPI_MAX_PORT_NAME	37
	MPI_VERSION	38
	MPI_SUBVERSION	39
	MPI_STATUS_SIZE (Fortran only)	40
	MPI_ADDRESS_KIND (Fortran only)	41
	MPI_COUNT_KIND (Fortran only)	42
	MPI_INTEGER_KIND (Fortran only)	43
	MPI_OFFSET_KIND (Fortran only)	44
	MPI_SUBARRAYS_SUPPORTED (Fortran only)	45
	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)	46
		47

 $^{24}$ 

	1	The constants that cannot be used in initialization expressions or assignments in For-
	2	tran are: MPI_BOTTOM
	3	MPI_STATUS_IGNORE
	4	MPI_STATUSES_IGNORE
	5	MPI_ERRCODES_IGNORE
	6	MPI_IN_PLACE
	7	MPI_ARGV_NULL
	8	MPI_ARGVS_NULL
	9	MPI_UNWEIGHTED
	10	MPI_WEIGHTS_EMPTY
	11	
	12	Advice to implementors. In Fortran the implementation of these special constants
	13	may require the use of language constructs that are outside the Fortran standard.
	14	Using special values for the constants (e.g., by defining them through PARAMETER
	15	statements) is not possible because an implementation cannot distinguish these val-
	16	ues from valid data. Typically, these constants are implemented as predefined static
	17	variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that
	18	the target compiler passes data by address. Inside the subroutine, this address can
	19	be extracted by some mechanism outside the Fortran standard (e.g., by Fortran ex- tensions on by implementing the function in $C$ ) (End of advice to implementate)
	20	tensions or by implementing the function in C). (End of advice to implementors.)
	21	
	22	2.5.5 Choice
	23	MPI functions sometimes use arguments with a <i>choice</i> (or union) data type. Distinct calls
	24	to the same routine may pass by reference actual arguments of different types. The mecha-
	25	nism for providing such arguments will differ from language to language. For Fortran with
ticketWG	26	the [include file mpif.h or the mpi module]mpi module or the deprecated mpif.h include
	27	file, the document uses <type> to represent a choice variable; with the Fortran mpi_f08</type>
	28	module, such arguments are declared with the Fortran 2008 + TR 29113 syntax TYPE(*),
	29	DIMENSION(); for C, we use void *.
	30	
	31	Advice to implementors. Implementors can freely choose how to implement choice

Advice to implementors. Implementors can freely choose how to implement choice arguments in the mpi module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the mpi\_f08 module. See details in Section 17.1.1 on page 601. (End of advice to implementors.)

# 2.5.6 Addresses

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.

45 46

32

33

34

35 36

37

# 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. 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, and ISO C, in particular. (Note that ANSI C has been replaced by ISO C.) 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 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, however, we expect that C programmers will understand the word "argument" (which has no specific meaning in C), thus allowing us to avoid unnecessary confusion for Fortran programmers.

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.

#### 2.6.1 Deprecated and Removed Names and Functions

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15 on page 594, but that users are recommended not to continue using, since better solutions were provided

 $\overline{7}$ 

with 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 C binding, and causes
 problems on machines with 32 bit INTEGERs and 64 bit addresses. In MPI-2, these functions
 were given new names with new bindings for the address arguments. The use of the old
 functions is deprecated. For consistency, here and in a few other cases, new C functions are
 also provided, even though the new functions are equivalent to the old functions. The old
 names are deprecated.

<sup>8</sup> Some of the deprecated constructs are now removed, as documented in Chapter 16 on <sup>9</sup> page 599. They may still be provided by an implementation for backwards compatibility, <sup>10</sup> but are not required.

Table 2.1 shows a list of all of the deprecated and removed constructs. Note that some
 C typedefs and Fortran subroutine names are included in this list; they are the types of
 callback functions.

14				
	Deprecated or removed	deprecated	removed	Replacement
15	construct	since	since	
16	MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
17	MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
18	MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
19	MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT
	MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
20	MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
21	MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
22	MPI_LB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
23	MPI_UB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
	MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLER
24	MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
25	MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
26	$MPI_{Handler_{function}^2}$	MPI-2.0	MPI-3.0	$MPI\_Comm\_errhandler\_function^2$
27	MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
28	MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
	MPI_DUP_FN <sup>3</sup>	MPI-2.0		MPI_COMM_DUP_FN <sup>3</sup>
29	MPI_NULL_COPY_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_COPY_FN <sup>3</sup>
30	MPI_NULL_DELETE_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_DELETE_FN <sup>3</sup>
31	MPI_Copy_function <sup>2</sup>	MPI-2.0		$MPI\_Comm\_copy\_attr\_function^2$
32	COPY_FUNCTION <sup>3</sup>	MPI-2.0		COMM_COPY_ATTR_FUNCTION <sup>3</sup>
	$MPI_Delete_function^2$	MPI-2.0		MPI_Comm_delete_attr_function <sup>2</sup>
33	DELETE_FUNCTION <sup>3</sup>	MPI-2.0		COMM_DELETE_ATTR_FUNCTION <sup>3</sup>
34	MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
35	MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
36	MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
37	MPI_COMBINER_HVECTOR_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HVECTOR <sup>4</sup>
	MPI_COMBINER_HINDEXED_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HINDEXED <sup>4</sup>
38	MPI_COMBINER_STRUCT_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_STRUCT <sup>4</sup>
39	MPI:	MPI-2.2	MPI-3.0	C language binding
40	[ticketWG.]mpif.h <sup>5</sup>	[ticketWG.]MPI-4.0		[ticketWG.]mpi or mpi_f08 module
41	<sup>1</sup> Predefined datatype.			
42	<sup>2</sup> Callback prototype definition.			
	<sup>3</sup> Predefined callback routine.			
43	<sup>4</sup> Constant.			
44	[ticketWG.] <sup>5</sup> Header file.			
45	Other entries are regular MPI routines.			
46				
47	Table 9.1. De	eprecated and Rem	oved cons	tructs
48		precated and Rem		

#### 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 names, e.g., for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix MPI\_. To avoid conflicting with the profiling interface, programs must also avoid subroutines and 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 as discussed in Section 17.2.9.

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.

The older MPI Fortran bindings (the deprecated mpif.h and use mpi) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 17.1.16.

### 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 names (identifiers), e.g., for variables, functions, constants, types, or macros, beginning with the prefix MPI\_. To support the profiling interface, programs must not declare functions with names beginning with the prefix PMPI\_.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.h.

Almost all C functions return an error code. The successful return code will be MPI\_SUCCESS, but failure return codes are implementation dependent.

Type declarations are provided for handles to each category of opaque objects.

Array arguments are indexed from zero.

Logical flags are integers with value 0 meaning "false" and a non-zero value meaning "true."

Choice arguments are pointers of type void \*.

Address arguments are of MPI defined type MPI\_Aint. File displacements are of type MPI\_Offset. MPI\_Aint is defined to be an integer of the size needed to hold any valid address on the target architecture. MPI\_Offset is defined to be an integer of the size needed to hold any valid file size on the target architecture.

 $\overline{7}$ 

 $^{31}$ 

<sub>24</sub> ticketWG.

#### 2.6.4Functions 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 17.2.4, 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 15style. The codes executed by each process need not be identical. The processes communicate 16via calls to MPI communication primitives. Typically, each process executes in its own 17address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI 19calls are used. The interaction of an MPI program with other possible means of commu-20nication, I/O, and process management is not specified. Unless otherwise stated in the 21specification of the standard, MPI places no requirements on the result of its interaction 22 with external mechanisms that provide similar or equivalent functionality. This includes, 23but is not limited to, interactions with external mechanisms for process control, shared and  $^{24}$ remote memory access, file system access and control, interprocess communication, process 25signaling, and terminal I/O. High quality implementations should strive to make the results 26of such interactions intuitive to users, and attempt to document restrictions where deemed 27necessary.

- 2829
- 30  $^{31}$

32 33 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.

3435 36

37

#### 2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received 38 correctly, and the user does not need to check for transmission errors, time-outs, or other 39 error conditions. In other words, MPI does not provide mechanisms for dealing with failures 40 in the communication system. If the MPI implementation is built on an unreliable underly-41 ing mechanism, then it is the job of the implementor of the MPI subsystem to insulate the 42user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, 43 such failures will be reflected as errors in the relevant communication call. Similarly, MPI 44itself provides no mechanisms for handling processor failures. 45

Of course, MPI programs may still be erroneous. A program error can occur when 46 an MPI call is made with an incorrect argument (non-existing destination in a send op-47eration, buffer too small in a receive operation, etc.). This type of error would occur in 48

1

2

3

4

56

7

8 9

10

11 12

13 14

any implementation. In addition, a *resource error* may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3.

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.

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

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

 $\frac{44}{45}$ 

46

47

are available. This is an important point in achieving portability across platforms that
 provide the same set of services.

3 4

5

2.9.1 Independence of Basic Runtime Routines

<sup>6</sup> MPI programs require that library routines that are part of the basic language environment

<sup>7</sup> (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).

```
<sup>14</sup> int rank;
```

```
<sup>15</sup> MPI_Init((void *)0, (void *)0);
```

```
<sup>16</sup> MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
if (rank == 0) printf("Starting program\n");
```

```
<sup>18</sup> MPI_Finalize();
```

<sup>20</sup> The corresponding Fortran 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).

```
<sup>25</sup>
<sub>26</sub> MPI_Comm_rank(MPI_COMM_WORLD, &rank);
<sub>27</sub> printf("Output from task rank %d\n", rank);
```

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

31 32

33

# 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.

43 44

45

# 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.

7

9

1516

17

18

19

 $^{24}$ 

25

26

38

40

41

```
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
  if (myrank == 0)
                       /* code for process zero */
                                                                                    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, 4546size and type of the send buffer are specified by the first three parameters of the send 47operation. The message sent will contain the 13 characters of this variable. In addition, 48 the send operation associates an *envelope* with the message. This envelope specifies the

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.

<sup>11</sup> The next sections describe the blocking send and receive operations. We discuss send, <sup>12</sup> receive, blocking communication semantics, type matching requirements, type conversion in <sup>13</sup> heterogeneous environments, and more general communication modes. Nonblocking com-<sup>14</sup> munication is addressed next, followed by probing and canceling a message, channel-like <sup>15</sup> constructs and send-receive operations, ending with a description of the "dummy" process, <sup>16</sup> MPI\_PROC\_NULL.

- 3.2 Blocking Send and Receive Operations
- 3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

26	IN	buf	initial address of send buffer (choice)
27	IN	count	number of elements in send buffer (non-negative inte-
28			ger)
29 30	IN	datatype	datatype of each send buffer element (handle)
31	IN	dest	rank of destination (integer)
32	IN	tag	message tag (integer)
33	IN	comm	communicator (handle)
34 35			
36	int MPI_		count, MPI_Datatype datatype, int dest,
37	int tag, MPI_Comm comm)		
38	MPI_Send	l(buf, count, datatype, de	st, tag, comm, ierror) BIND(C)
39	TYPE(*), DIMENSION(), INTENT(IN) :: buf		

INTEGER, INTENT(IN) :: count, dest, tag

TYPE(MPI\_Datatype), INTENT(IN) :: datatype

TYPE(MPI\_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```
<sup>44</sup>
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<br/>(type> BUF(*)<br/>INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
<sup>47</sup> The blocking semantics of this call are described in Section 3.4.
```

17 18 19

20

21

22 23 24

25

40

41

42

#### 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 communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

The datatypes MPI\_AINT, MPI\_OFFSET, and MPI\_COUNT correspond to the MPIdefined C types MPI\_Aint, MPI\_Offset, and MPI\_Count and their Fortran equivalents  $\overline{7}$ 

 $^{31}$ 

 $46 \\ 47$ 

1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
4	MPI_SHORT	signed short int
5	MPI_INT	signed int
6	MPI_LONG	signed long int
7	MPI_LONG_LONG_INT	signed long long int
8	MPI_LONG_LONG (as a synonym)	signed long long int
9	MPI_SIGNED_CHAR	signed char
10		(treated as integral value)
11	MPI_UNSIGNED_CHAR	unsigned char
12		(treated as integral value)
13	MPI_UNSIGNED_SHORT	unsigned short int
14	MPI_UNSIGNED	unsigned int
15	MPI_UNSIGNED_LONG	unsigned long int
16	MPI_UNSIGNED_LONG_LONG	unsigned long long int
17	MPI_FLOAT	float
18	MPI_DOUBLE	double
19	MPI_LONG_DOUBLE	long double
20	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
24	MPI_INT8_T	int8_t
25	MPI_INT16_T	int16_t
26	MPI_INT32_T	int32_t
27	MPI_INT64_T	int64_t
28	MPI_UINT8_T	uint8_t
29	MPI_UINT16_T	uint16_t
30	MPI_UINT32_T	uint32_t
31	MPI_UINT64_T	uint64_t
32	MPI_C_COMPLEX	float _Complex
33	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38		
39	Table 3.2: Predefined MPI datatypes co	presponding to C datatypes
40	Table 5.2. I recentled with datatypes of	mosponding to C datatypes

INTEGER (KIND=MPI\_ADDRESS\_KIND), INTEGER (KIND=MPI\_OFFSET\_KIND), and INTEGER (KIND=MPI\_COUNT\_KIND). This is described in Table 3.3. All predefined datatype handles are available in all language bindings. See Sections 17.2.6 and 17.2.10 on page 654 and 662 for information on interlanguage communication with these types.

<sup>46</sup> If there is an accompanying C++ compiler then the datatypes in Table 3.4 are also <sup>47</sup> supported in C and Fortran.

MPI datatype	C datatype	Fortran datatype		
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)		
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)		
MPI_COUNT	MPI_Count	INTEGER (KIND=MPI_COUNT_KIND)		

Table 3.3: Predefined MPI datatypes corresponding to both C and Fortran datatypes

MPI datatype	C++ datatype
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	std::complex <float></float>
MPI_CXX_DOUBLE_COMPLEX	std::complex <double></double>
MPI_CXX_LONG_DOUBLE_COMPLEX	std::complex <long double=""></long>

Table 3.4: Predefined MPI datatypes corresponding to C++ datatypes

### 3.2.3 Message Envelope

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
$\operatorname{tag}$
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  $0, \ldots, UB$ , 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  $0, \ldots, n-1 \cup \{\text{MPI}\_\text{PROC}\_\text{NULL}\}$ , where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

A predefined communicator MPI\_COMM\_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI\_COMM\_WORLD.

 $^{31}$ 

 $45 \\ 46$ 

1 Advice to users. Users that are comfortable with the notion of a flat name space  $\mathbf{2}$ for processes, and a single communication context, as offered by most existing com-3 munication libraries, need only use the predefined variable MPI\_COMM\_WORLD as the 4 comm argument. This will allow communication with all the processes available at 5initialization time. 6 Users may define new communicators, as explained in Chapter 6. Communicators 7 provide an important encapsulation mechanism for libraries and modules. They allow 8 modules to have their own disjoint communication universe and their own process 9 numbering scheme. (End of advice to users.) 10 11 Advice to implementors. The message envelope would normally be encoded by a 12fixed-length message header. However, the actual encoding is implementation depen-13 dent. Some of the information (e.g., source or destination) may be implicit, and need 14not be explicitly carried by messages. Also, processes may be identified by relative 15ranks, or absolute ids, etc. (End of advice to implementors.) 16173.2.4 **Blocking Receive** 18 The syntax of the blocking receive operation is given below. 192021MPI\_RECV (buf, count, datatype, source, tag, comm, status) 22 OUT 23buf initial address of receive buffer (choice)  $^{24}$ IN count number of elements in receive buffer (non-negative in-25teger) 26IN datatype datatype of each receive buffer element (handle) 2728IN rank of source or MPI\_ANY\_SOURCE (integer) source 29message tag or MPI\_ANY\_TAG (integer) IN tag 30 IN comm communicator (handle)  $^{31}$ 32 OUT status status object (Status) 33 34 int MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source, 35 int tag, MPI\_Comm comm, MPI\_Status \*status) 36 MPI\_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C) 37 TYPE(\*), DIMENSION(..) :: buf 38 INTEGER, INTENT(IN) :: count, source, tag 39 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 40 TYPE(MPI\_Comm), INTENT(IN) :: comm 41 TYPE(MPI\_Status) :: status 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44MPI\_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) 45<type> BUF(\*) 46INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), 47 IERROR 48

#### 3.2. BLOCKING SEND AND RECEIVE OPERATIONS

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

The receive buffer consists of the storage containing **count** consecutive elements of the type specified by datatype, starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

Advice to users. The MPI\_PROBE function described in Section 3.8 can be used to receive messages of unknown length. (End of advice to users.)

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive 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.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow 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 it is an odd address. (End of advice to implementors.)

The selection of a message by a receive operation is governed by the value of the 23message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI\_ANY\_SOURCE value for source, and/or a wildcard MPI\_ANY\_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a 27wildcard 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 2930 source unless source=MPI\_ANY\_SOURCE in the pattern, and has a matching tag unless tag=MPI\_ANY\_TAG in the pattern.

The message tag is specified by the tag argument of the receive operation. The argu-33 ment source, if different from MPI\_ANY\_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommu-34 35 nicators). Thus, the range of valid values for the source argument is  $\{0, \ldots, n-1\} \cup$  $\{MPI_ANY_SOURCE\}, \cup \{MPI_PROC_NULL\}, where n is the number of processes in this group.$ 36

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify a unique receiver. This matches a "push" communication mechanism, where data transfer is effected by the sender (rather than a "pull" mechanism, where data transfer is effected by 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 46be implemented as an additional tag field. It differs from the regular message tag 47in that wild card matching is not allowed on this field, and that value setting for 48

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10 11

12

13

14

15

1617

18

19

20

21

22

 $^{24}$ 

2526

28

 $^{31}$ 

32

37

38 39

40

41 42

43

this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

The use of dest or source=MPI\_PROC\_NULL to define a "dummy" destination or source in any send or receive call is described in Section 3.11 on page 80.

## 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,

and MPI\_ERROR, the structure may contain additional nerds. Thus,
 status.MPI\_SOURCE, status.MPI\_TAG and status.MPI\_ERROR contain the source, tag, and
 error code, respectively, of the received message.

In Fortran with USE mpi or the deprecated 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 17.2.5 on page 652.

*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.*)

*Rationale.* It is allowed to have the same name (e.g., MPI\_SOURCE) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

<sup>39</sup> In general, message-passing calls do not modify the value of the error code field of <sup>40</sup> status variables. This field may be updated only by the functions in Section 3.7.5 which <sup>41</sup> return multiple statuses. The field is updated if and only if such function returns with an <sup>42</sup> 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.)

1

2

3 4

5 6 7

8

9

10

11

12

13

14

15

32

33

34

35 36

37

38

ticketWG.  $^{18}_{19}$ 

### 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)

MPI\_Get\_count(status, datatype, count, ierror) BIND(C)
 TYPE(MPI\_Status), INTENT(IN) :: status
 TYPE(MPI\_Datatype), INTENT(IN) :: datatype
 INTEGER, INTENT(OUT) :: count
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# MPI\_GET\_COUNT(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, COUNT, IERROR

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. 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 transferred 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

 $^{24}$ 

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.

111213

21

22

23

4748

1

2

3

4

5

6

7 8

9

10

#### Passing MPI\_STATUS\_IGNORE for Status 3.2.6

14Every call to MPI\_RECV includes a status argument, wherein the system can return details 15about the message received. There are also a number of other MPI calls where status is 16 returned. An object of type MPI\_Status is not an MPI opaque object; its structure is declared in mpi.h and the deprecated mpif.h, and it exists in the user's program. In many cases, application programs are constructed so that it is unnecessary for them to examine 19the status fields. In these cases, it is a waste for the user to allocate a status object, and 20it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE, which when passed to a receive, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that

 $^{24}$ MPI\_STATUS\_IGNORE is not a special type of MPI\_Status object; rather, it is a special value 25for the argument. In C one would expect it to be NULL, not the address of a special 26MPI\_Status. 27

MPI\_STATUS\_IGNORE, and the array version MPI\_STATUSES\_IGNORE, can be used every-28where a status argument is passed to a receive, wait, or test function. MPI\_STATUS\_IGNORE 29cannot be used when status is an IN argument. Note that in Fortran MPI\_STATUS\_IGNORE 30 and MPI\_STATUSES\_IGNORE are objects like MPI\_BOTTOM (not usable for initialization or  $^{31}$ assignment). See Section 2.5.4. 32

In general, this optimization can apply to all functions for which status or an array of 33 statuses is an OUT argument. Note that this converts status into an INOUT argument. The 34functions that can be passed MPI\_STATUS\_IGNORE are all the various forms of MPI\_RECV, 35 MPI\_PROBE, MPI\_TEST, and MPI\_WAIT, as well as MPI\_REQUEST\_GET\_STATUS. When 36 an array is passed, as in the MPI\_{TEST|WAIT}{ALL|SOME} functions, a separate constant, 37 MPI\_STATUSES\_IGNORE, is passed for the array argument. It is possible for an MPI function 38 to return MPI\_ERR\_IN\_STATUS even when MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE 39 has been passed to that function. 40

MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE are not required to have the same 41 values in C and Fortran. 42

It is not allowed to have some of the statuses in an array of statuses for 43 MPI\_{TEST|WAIT}{ALL|SOME} functions set to MPI\_STATUS\_IGNORE; one either specifies 44ignoring all of the statuses in such a call with MPI\_STATUSES\_IGNORE, or *none* of them by 45passing normal statuses in all positions in the array of statuses. 46

17ticket WG.  $_{18}$ 

#### 3.3 Data Type Matching and Data Conversion

### 3.3.1 Type Matching Rules

One can think of message transfer as consisting of the following three phases.

- 1. Data is pulled out of the send buffer and a message is assembled.
- 2. A message is transferred from sender to receiver.
- 3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

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

 $^{24}$ The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic 26type of the host program variable. For example, an entry with type name MPI\_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran 2728and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI\_BYTE or MPI\_PACKED can be used to match any byte of storage (on a 2930 byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI\_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type MPI\_BYTE allows one to transfer the binary value of a byte in memory unchanged.

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 3.1 Sender and receiver specify matching types. 1 2

5

6 7

8 9

10 11

12

13

1415

16

17

18

19 20

21

22

23

25

 $^{31}$ 

32 33

34

35

36

37

38 39

40

41

42

43

4445

4647

```
1
     CALL MPI_COMM_RANK(comm, rank, ierr)
\mathbf{2}
     IF (rank.EQ.0) THEN
3
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
4
     ELSE IF (rank.EQ.1) THEN
\mathbf{5}
          CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
6
     END IF
7
          This code is correct if both a and b are real arrays of size \geq 10. (In Fortran, it might be
8
     correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
9
     to an array with ten reals.)
10
11
                      Sender and receiver do not specify matching types.
     Example 3.2
12
13
     CALL MPI_COMM_RANK(comm, rank, ierr)
14
     IF (rank.EQ.0) THEN
15
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
16
     ELSE IF (rank.EQ.1) THEN
17
          CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
18
     END IF
19
          This code is erroneous, since sender and receiver do not provide matching datatype
20
     arguments.
21
22
     Example 3.3
                      Sender and receiver specify communication of untyped values.
23
24
     CALL MPI_COMM_RANK(comm, rank, ierr)
25
     IF (rank.EQ.0) THEN
26
          CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
27
     ELSE IF (rank.EQ.1) THEN
28
          CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
29
     END IF
30
          This code is correct, irrespective of the type and size of a and b (unless this results in
^{31}
     an out of bounds memory access).
32
33
           Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND,
34
           then MPI will send the data stored at contiguous locations, starting from the address
35
           indicated by the buf argument. This may have unexpected results when the data
36
           layout is not as a casual user would expect it to be. For example, some Fortran
37
           compilers implement variables of type CHARACTER as a structure that contains the
38
           character length and a pointer to the actual string. In such an environment, sending
39
           and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have
40
           the anticipated result of transferring the character string. For this reason, the user is
41
           advised to use typed communications whenever possible. (End of advice to users.)
42
43
     Type MPI_CHARACTER
44
45
     The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER,
46
     rather than the entire character string stored in the variable. Fortran variables of type
47
     CHARACTER or substrings are transferred as if they were arrays of characters. This is
```

<sup>48</sup> illustrated in the example below.

Example 3.4
Transfer of Fortran CHARACTERs.
CHARACTER*10 a
CHARACTER*10 b
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF

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.

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

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.*)

### 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.

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. <sup>47</sup> On the other hand, MPI requires that a representation conversion be performed when a <sup>48</sup>

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 and character values, and to convert a
 floating point value to the nearest value that can be represented on the target system.

<sup>5</sup> Overflow and underflow exceptions may occur during floating point conversions. Con-<sup>6</sup> version of integers or characters may also lead to exceptions when a value that can be <sup>7</sup> represented in one system cannot be represented in the other system. An exception occur-<sup>8</sup> ring during representation conversion results in a failure of the communication. An error <sup>9</sup> occurs either in the send operation, or the receive operation, or both.

<sup>10</sup> If a value sent in a message is untyped (i.e., of type MPI\_BYTE), then the binary <sup>11</sup> representation of the byte stored at the receiver is identical to the binary representation <sup>12</sup> of the byte loaded at the sender. This holds true, whether sender and receiver run in the <sup>13</sup> same or in distinct environments. No representation conversion is required. (Note that <sup>14</sup> representation conversion may occur when values of type MPI\_CHARACTER or MPI\_CHAR <sup>15</sup> are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

<sup>16</sup> No conversion need occur when an MPI program executes in a homogeneous system,
 <sup>17</sup> where all processes run in the same environment.

18 Consider the three examples, 3.1-3.3. The first program is correct, assuming that **a** and 19b are REAL arrays of size > 10. If the sender and receiver execute in different environments, 20then the ten real values that are fetched from the send buffer will be converted to the 21representation for reals on the receiver site before they are stored in the receive buffer. 22 While the number of real elements fetched from the send buffer equal the number of real 23elements stored in the receive buffer, the number of bytes stored need not equal the number  $^{24}$ of bytes loaded. For example, the sender may use a four byte representation and the receiver 25an eight byte representation for reals.

26

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.

<sup>34</sup> Data representation conversion also applies to the envelope of a message: source, des-<sup>35</sup> tination 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.*)

44 45

36

37

38

39

40

41 42

43

<sup>46</sup> MPI requires support for inter-language communication, i.e., if messages are sent by a
 <sup>47</sup> C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined
 <sup>48</sup> in Section 17.2 on page 649.

## 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.

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.

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

There are three additional communication modes.

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

A send that uses the *synchronous* mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its  $\frac{44}{45}$ 

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

execution, namely that it has started executing the matching receive. If both sends and
 receives are blocking operations then the use of the synchronous mode provides synchronous
 communication semantics: a communication does not complete at either end before both
 processes rendezvous at the communication. A send executed in this mode is *non-local*.

5A send that uses the *ready* communication mode may be started *only* if the matching 6 receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-7fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 8 required and results in improved performance. The completion of the send operation does 9 not depend on the status of a matching receive, and merely indicates that the send buffer 10 can be reused. A send operation that uses the ready mode has the same semantics as a 11standard send operation, or a synchronous send operation; it is merely that the sender 12provides additional information to the system (namely that a matching receive is already 13posted), that can save some overhead. In a correct program, therefore, a ready send could 14be replaced by a standard send with no effect on the behavior of the program other than 15performance.

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.

19 20

21

MPI\_BSEND (buf, count, datatype, dest, tag, comm)

22IN buf initial address of send buffer (choice) 23IN count number of elements in send buffer (non-negative inte- $^{24}$ ger) 25datatype of each send buffer element (handle) IN datatype 2627IN dest rank of destination (integer) 28IN message tag (integer) tag 29 IN comm communicator (handle) 30  $^{31}$ int MPI\_Bsend(const void\* buf, int count, MPI\_Datatype datatype, int dest, 32 33 int tag, MPI\_Comm comm) 34MPI\_Bsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C) 35 TYPE(\*), DIMENSION(..), INTENT(IN) :: buf 36 INTEGER, INTENT(IN) :: count, dest, tag 37 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 38 TYPE(MPI\_Comm), INTENT(IN) :: comm 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40  $^{41}$ MPI\_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 42<type> BUF(\*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 43 44 Send in buffered mode. 4546 47 48

MPI_	SSEND (buf, count, datatype, dest, <sup>.</sup>	tag, comm)	1		
IN	buf	initial address of send buffer (choice)	2		
IN	count	number of elements in send buffer (non-negative inte-	3 4		
		ger)	5		
IN	datatype	datatype of each send buffer element (handle)	6		
IN	dest	rank of destination (integer)	7		
IN		message tag (integer)	8		
	tag	, - ,	9 10		
IN	comm	communicator (handle)	10		
<pre>int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>					
MPI_S	Ssend(buf, count, datatype, de	est, tag, comm, ierror) BIND(C)	15		
	TYPE(*), DIMENSION(), INTENT		16		
	INTEGER, INTENT(IN) :: count,		17		
	<pre>TYPE(MPI_Datatype), INTENT(IN) TYPE(MPI_Comm), INTENT(IN) ::</pre>	v1	18		
	INTEGER, OPTIONAL, INTENT(IN) ::		19 20		
			20		
	SSEND(BUF, COUNT, DATATYPE, DE	ST, TAG, CUMM, IERROR)	22		
	<pre><type> BUF(*) INTEGER COUNT, DATATYPE, DEST,</type></pre>	TAC COMM TERROR	23		
			24		
S	send in synchronous mode.		25		
			26 27		
MPI_	RSEND (buf, count, datatype, dest,	tag, comm)	27		
IN	buf	initial address of send buffer (choice)	29		
IN	count	number of elements in send buffer (non-negative inte-	30		
		ger)	31		
IN	datatype	datatype of each send buffer element (handle)	32		
IN		rank of destination (integer)	33 34		
	dest		35		
IN	tag	message tag (integer)	36		
IN	comm	communicator (handle)	37		
			38		
int M		count, MPI_Datatype datatype, int dest,	39		
	int tag, MPI_Comm cor		40		
		est, tag, comm, ierror) BIND(C)	41 42		
TYPE(*), DIMENSION(), INTENT(IN) :: buf					
INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm					
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)					

1	
2	<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR</type>
3	INTEGER COONT, DATATIFE, DEST, TAG, COMM, TERROR
4	Send in ready mode.
5	There is only one receive operation, but it matches any of the send modes. The receive
6	operation described in the last section is <i>blocking</i> : it returns only after the receive buffer
7	contains the newly received message. A receive can complete before the matching send has
8	completed (of course, it can complete only after the matching send has started).
9	In a multithreaded 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
10	the same address space. In such a case it is the user's responsibility not to modify a
11	communication buffer until the communication completes. Otherwise, the outcome of the
12	computation is undefined.
13 14	1
15	Advice to implementors. Since a synchronous send cannot complete before a matching
16	receive is posted, one will not normally buffer messages sent by such an operation.
17	It is recommended to choose buffering over blocking the sender, whenever possible,
18	for standard sends. The programmer can signal his or her preference for blocking the
19	sender until a matching receive occurs by using the synchronous send mode.
20	A possible communication protocol for the various communication modes is outlined
21	below.
22 23	ready send: The message is sent as soon as possible.
23 24	synchronous send: The sender sends a request-to-send message. The receiver stores
25	this request. When a matching receive is posted, the receiver sends back a permission-
26	to-send message, and the sender now sends the message.
27	standard send: First protocol may be used for short messages, and second protocol
28	for long messages.
29	buffered send: The sender copies the message into a buffer and then sends it with a
30 31	nonblocking send (using the same protocol as for standard send).
32	Additional control messages might be needed for flow control and error recovery. Of
33	course, there are many other possible protocols.
34	Ready send can be implemented as a standard send. In this case there will be no
35	performance advantage (or disadvantage) for the use of ready send.
36	A standard send can be implemented as a synchronous send. In such a case, no data
37	buffering is needed. However, users may expect some buffering.
38 30	In a multithreaded environment, the execution of a blocking communication should
39 40	block only the executing thread, allowing the thread scheduler to de-schedule this
41	thread and schedule another thread for execution. (End of advice to implementors.)
42	
43	3.5 Semantics of Point-to-Point Communication
44	
45 46	A valid MPI implementation guarantees certain general properties of point-to-point com-

Order Messages are *non-overtaking*: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends 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 calls described later, such as MPI\_CANCEL or MPI\_WAITANY, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multithreaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically 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 successively sent messages, then the two messages can match the two receives in either order.

**Example 3.5** An example of non-overtaking messages.

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

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

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

**Example 3.6** An example of two, intertwined matching pairs.

```
40
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                   41
IF (rank.EQ.0) THEN
                                                                                   42
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
                                                                                   43
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
                                                                                   44
ELSE IF (rank.EQ.1) THEN
                                                                                   45
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
                                                                                   46
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
                                                                                   47
END IF
                                                                                   48
```

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

1 Both processes invoke their first communication call. Since the first send of process zero  $\mathbf{2}$ uses the buffered mode, it must complete, irrespective of the state of process one. Since 3 no matching receive is posted, the message will be copied into buffer space. (If insufficient 4 buffer space is available, then the program will fail.) The second send is then invoked. At  $\mathbf{5}$ that point, a matching pair of send and receive operation is enabled, and both operations 6 must complete. Process one next invokes its second receive call, which will be satisfied by  $\overline{7}$ the buffered message. Note that process one received the messages in the reverse order they 8 were sent.

10Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose 11that a send is posted. Then it is possible that the destination process repeatedly posts a 12receive that matches this send, yet the message is never received, because it is each time 13overtaken by another message, sent from another source. Similarly, suppose that a receive 14was posted by a multithreaded process. Then it is possible that messages that match this 15receive are repeatedly received, yet the receive is never satisfied, because it is overtaken 16by other receives posted at this node (by other executing threads). It is the programmer's 17responsibility to prevent starvation in such situations. 18

19Resource limitations Any pending communication operation consumes system resources 20that are limited. Errors may occur when lack of resources prevent the execution of an MPI 21call. A quality implementation will use a (small) fixed amount of resources for each pending 22send in the ready or synchronous mode and for each pending receive. However, buffer space 23may be consumed to store messages sent in standard mode, and must be consumed to store  $^{24}$ messages sent in buffered mode, when no matching receive is available. The amount of space 25available for buffering will be much smaller than program data memory on many systems. 26Then, it will be easy to write programs that overrun available buffer space.

<sup>27</sup> MPI allows the user to provide buffer memory for messages sent in the buffered mode. <sup>28</sup> Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI <sup>30</sup> implementation is required to do no worse than implied by this model. This allows users to <sup>31</sup> avoid buffer overflows when they use buffered sends. Buffer allocation and use is described <sup>31</sup> in Section 3.6.

32 A buffered send operation that cannot complete because of a lack of buffer space is 33 erroneous. When such a situation is detected, an error is signaled that may cause the 34program to terminate abnormally. On the other hand, a standard send operation that 35 cannot complete because of lack of buffer space will merely block, waiting for buffer space 36 to become available or for a matching receive to be posted. This behavior is preferable in 37 many situations. Consider a situation where a producer repeatedly produces new values 38and sends them to a consumer. Assume that the producer produces new values faster 39 than the consumer can consume them. If buffered sends are used, then a buffer overflow 40will result. Additional synchronization has to be added to the program so as to prevent 41 this from occurring. If standard sends are used, then the producer will be automatically 42throttled, as its send operations will block when buffer space is unavailable. 43

<sup>43</sup> In some situations, a lack of buffer space leads to deadlock situations. This is illustrated <sup>44</sup> by the examples below.

<sup>46</sup> **Example 3.7** An exchange of messages.

42

9

ELSE IF (rank.EQ.1) THEN

END IF

```
1
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                        2
IF (rank.EQ.0) THEN
                                                                                        3
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
                                                                                        4
ELSE IF (rank.EQ.1) THEN
                                                                                        5
                                                                                        6
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
                                                                                        8
                                                                                        9
This program will succeed even if no buffer space for data is available. The standard send
                                                                                        10
operation can be replaced, in this example, with a synchronous send.
                                                                                        11
                                                                                        12
Example 3.8
               An errant attempt to exchange messages.
                                                                                        13
                                                                                        14
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                        15
IF (rank.EQ.0) THEN
                                                                                        16
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
                                                                                        17
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
                                                                                        18
ELSE IF (rank.EQ.1) THEN
                                                                                       19
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
                                                                                       20
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
                                                                                       21
END IF
                                                                                       22
The receive operation of the first process must complete before its send, and can complete
                                                                                       23
only if the matching send of the second processor is executed. The receive operation of the
                                                                                        24
second process must complete before its send and can complete only if the matching send
                                                                                        25
of the first process is executed. This program will always deadlock. The same holds for any
                                                                                        26
other send mode.
                                                                                       27
                                                                                       28
Example 3.9
               An exchange that relies on buffering.
                                                                                       29
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                        30
IF (rank.EQ.0) THEN
                                                                                        31
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
                                                                                        32
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
                                                                                       33
```

Advice to users.When standard send operations are used, then a deadlock situation43may occur where both processes are blocked because buffer space is not available.44same will certainly happen, if the synchronous mode is used.1f the buffered mode isused, and not enough buffer space is available, then the program will not complete46either.However, rather than a deadlock situation, we shall have a buffer overflow47error.48

CALL MPI\_SEND(sendbuf, count, MPI\_REAL, 0, tag, comm, ierr)

communication system can buffer at least count words of data.

CALL MPI\_RECV(recvbuf, count, MPI\_REAL, 0, tag, comm, status, ierr)

The message sent by each process has to be copied out before the send operation returns

and the receive operation starts. For the program to complete, it is necessary that at least

one of the two messages sent be buffered. Thus, this program can succeed only if the

34

35

36

37 38

39

40

41

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 3.9. 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 10 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 12might also be used for debugging purposes, as buffer overflow conditions are easier to 13 diagnose than deadlock conditions. 14

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.)

19 2021

22 23

 $^{24}$ 

2526

2728

29

30  $^{31}$ 

32 33

34

35

36

37

38

39

40

4748

#### 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.

```
MPI_BUFFER_ATTACH(buffer, size)
```

```
IN
           buffer
                                      initial buffer address (choice)
  IN
           size
                                      buffer size, in bytes (non-negative integer)
int MPI_Buffer_attach(void* buffer, int size)
MPI_Buffer_attach(buffer, size, ierror) BIND(C)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                          ierror
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
```

41 Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes-42sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be 43 attached to a process at a time. In C, buffer is the starting address of a memory region. In 44Fortran, one can pass the first element of a memory region or a whole array, which must be 45'simply contiguous' (for 'simply contiguous,' see also Section 17.1.12 on page 630. 46

44

1

2

3

4

5

6

7

8

9

11

15

16

17

associated with the process.

MPI\_BUFFER\_DETACH(buffer\_addr, size) 1  $\mathbf{2}$ OUT buffer\_addr initial buffer address (choice) 3 OUT size buffer size, in bytes (non-negative integer) 4 5 int MPI\_Buffer\_detach(void\* buffer\_addr, int\* size) 6 7 MPI\_Buffer\_detach(buffer\_addr, size, ierror) BIND(C) 8 USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 9 TYPE(C\_PTR), INTENT(OUT) :: buffer\_addr 10 INTEGER, INTENT(OUT) :: size 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12MPI\_BUFFER\_DETACH(BUFFER\_ADDR, SIZE, IERROR) 13 <type> BUFFER\_ADDR(\*) 14INTEGER SIZE, IERROR 1516Detach the buffer currently associated with MPI. The call returns the address and the 17 size of the detached buffer. This operation will block until all messages currently in the 18 buffer have been transmitted. Upon return of this function, the user may reuse or deallocate 19 the space taken by the buffer. 2021**Example 3.10** Calls to attach and detach buffers. 22 #define BUFFSIZE 10000 23int size;  $^{24}$ char \*buff; 25MPI\_Buffer\_attach( malloc(BUFFSIZE), BUFFSIZE); 26/\* a buffer of 10000 bytes can now be used by MPI\_Bsend \*/ 27MPI\_Buffer\_detach( &buff, &size); 28/\* Buffer size reduced to zero \*/ 29 MPI\_Buffer\_attach( buff, size); 30 /\* Buffer of 10000 bytes available again \*/ 3132 Even though the C functions MPI\_Buffer\_attach and Advice to users. 33 MPI\_Buffer\_detach both have a first argument of type void\*, these arguments are used 34differently: A pointer to the buffer is passed to MPI\_Buffer\_attach; the address of the 35 pointer is passed to MPI\_Buffer\_detach, so that this call can return the pointer value. <sup>36</sup> ticketWG. In Fortran with the mpi module or the deprecated mpif.h, the type of the buffer\_addr 37 argument is wrongly defined and the argument is therefore unused. In Fortran with 38the mpi\_f08 module, the address of the buffer is returned as TYPE(C\_PTR), see also 39 Example 8.1 on page 339 about the use of C\_PTR pointers. (End of advice to users.) 40 Rationale. Both arguments are defined to be of type void\* (rather than 41 void\* and void\*\*, respectively), so as to avoid complex type casts. E.g., in the last 42example, &buff, which is of type char\*\*, can be passed as argument to 43 MPI\_Buffer\_detach without type casting. If the formal parameter had type void\*\* 44 then we would need a type cast before and after the call. (*End of rationale.*) 4546The statements made in this section describe the behavior of MPI for buffered-mode 47sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is

<sup>1</sup> MPI must provide as much buffering for outgoing messages *as if* outgoing message <sup>2</sup> data were buffered by the sending process, in the specified buffer space, using a circular, <sup>3</sup> contiguous-space allocation policy. We outline below a model implementation that defines <sup>4</sup> this policy. MPI may provide more buffering, and may use a better buffer allocation algo-<sup>5</sup> rithm than described below. On the other hand, MPI may signal an error whenever the <sup>6</sup> simple buffering allocator described below would run out of space. In particular, if no buffer <sup>7</sup> 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.

30 31

32

33

34

35

36

37

38

39

40

41 42

43

44

45 46

47

48

12

13

14

15 16

17

18 19

20 21

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.
- 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

#### 3.7 Nonblocking Communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use *nonblocking communication*. A nonblocking *send start* call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer. A separate *receive complete* call is needed to complete the receive operation and verify that 20the data has been received into the receive buffer. With suitable hardware, the transfer 21of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also 22avoid system buffering and memory-to-memory copying, as information is provided early 23on the location of the receive buffer.

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

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 receive-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

1

2

3 4 5

6  $\overline{7}$ 

8 9

10 11

12

13

1415

16

17

18

19

 $^{24}$ 

25

26

2728

2930

31

32 33

34

35

36

37

38 39

40 41

42

43

44

4546

47

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.*)

#### 23 24 25

## 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.

## 3.7.2 Communication Initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for *buffered*, *synchronous* or *ready* mode. In addition a prefix of I (for *immediate*) indicates that the call is nonblocking.

38 39

34

35

36

37

- 40 41
- 42

43

- 44
- 45
- 46

47

48

1

2 3

4

5 6

7 8

9

10

11

12

13

14

15

16

17

18

19

20

21

### 3.7. NONBLOCKING COMMUNICATION

MPI_ISEND(buf, count, datatype, dest, tag, comm, request) <sup>1</sup>			
IN	buf	initial address of send buffer (choice)	2 3
IN	count	number of elements in send buffer (non-negative inte-	4 5
IN	datatype		6
IN	dest		7
IN	tag		8 9
IN	comm		9 10
OUT	request		11
001	request	- 、 / 1	12
int MPI_1		t count, MPI_Datatype datatype, int dest,	13 14 15
TYPE INTEC TYPE TYPE		<pre>rest, tag, comm, request, reffor) BIND(c) T(IN), ASYNCHRONOUS :: buf , dest, tag ) :: datatype     comm 2 ) :: request ) :: ierror 2 </pre>	16 17 18 19 20 21 22
<type INTEC</type 	e> BUF(*)	EST, TAG, COMM, REQUEST, IERROR)	23 24 25 26 27 28 29
MPI_IBSE	ND(buf, count, datatype, dest,		29 30
IN	buf	initial address of send buffer (choice)	31
IN	count	number of elements in send buffer (non-negative inte-	32 33 34
IN	datatype		35
IN	dest	rank of destination (integer)	36
IN	tag	message tag (integer)	37 38
IN	comm		39
OUT	request	communication request (handle)	40
	int tag, MPI_Comm co	nt count, MPI_Datatype datatype, int dest, mm, MPI_Request *request)	41 42 43 44
<pre>MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C) 45 TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf 46 INTEGER, INTENT(IN) :: count, dest, tag 47 TYPE(MPI_Datatype), INTENT(IN) :: datatype 48</pre>			

```
1
          TYPE(MPI_Comm), INTENT(IN) :: comm
\mathbf{2}
          TYPE(MPI_Request), INTENT(OUT) ::
                                                 request
3
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
4
     MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
5
          <type> BUF(*)
6
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
7
8
         Start a buffered mode, nonblocking send.
9
10
     MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
11
12
       IN
                 buf
                                             initial address of send buffer (choice)
13
       IN
                                             number of elements in send buffer (non-negative inte-
                 count
14
                                             ger)
15
                                             datatype of each send buffer element (handle)
       IN
                 datatype
16
17
       IN
                 dest
                                             rank of destination (integer)
18
       IN
                 tag
                                             message tag (integer)
19
       IN
                 comm
                                             communicator (handle)
20
21
       OUT
                 request
                                             communication request (handle)
22
23
     int MPI_Issend(const void* buf, int count, MPI_Datatype datatype, int dest,
^{24}
                    int tag, MPI_Comm comm, MPI_Request *request)
25
26
     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
          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) ::
^{31}
                                                 request
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
32
33
     MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
34
          <type> BUF(*)
35
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
36
37
         Start a synchronous mode, nonblocking send.
38
39
40
41
42
43
44
45
46
47
48
```

MPI_IRSE	ND(buf, count, datatype, dest,	tag, comm, request)	1	
IN	buf	initial address of send buffer (choice)	2 3	
IN	count	number of elements in send buffer (non-negative integer)	4 5	
IN	datatype	datatype of each send buffer element (handle)	6	
IN	dest	rank of destination (integer)	7	
IN	tag	message tag (integer)	8 9	
IN	comm	communicator (handle)	10	
OUT	request	communication request (handle)	11 12	
int MPI_		nt count, MPI_Datatype datatype, int dest, mm, MPI_Request *request)	13 14 15	
TYPE INTE TYPE TYPE TYPE		) :: datatype comm ) :: request	16 17 18 19 20 21 22 23	
<type INTE</type 	e> BUF(*)	DEST, TAG, COMM, REQUEST, IERROR) , TAG, COMM, REQUEST, IERROR nd.	24 25 26 27 28	
MPI_IREC	V (buf, count, datatype, source	e, tag, comm, request)	29 30	
OUT	buf	initial address of receive buffer (choice)	31	
IN	count	number of elements in receive buffer (non-negative in- teger)	32 33 34	
IN	datatype	datatype of each receive buffer element (handle)	35	
IN	source	rank of source or MPI_ANY_SOURCE (integer)	36	
IN	tag	message tag or MPI_ANY_TAG (integer)	37 38	
IN	comm	communicator (handle)	39	
OUT	request	communication request (handle)	40 41	
	int tag, MPI_Comm co	t, MPI_Datatype datatype, int source, mm, MPI_Request *request)	42 43 44	
TYPE INTEC	<pre>MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND(C) 45 TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, source, tag 47 TYPE(VEL Detect) INTENT(IN)</pre>			
TYPE	(MPI_Datatype), INTENT(IN	) :: datatype	48	

1 TYPE(MPI\_Comm), INTENT(IN) :: comm 2 TYPE(MPI\_Request), INTENT(OUT) :: request 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI\_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 5<type> BUF(\*) 6 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 7 8 Start a nonblocking receive. 9 These calls allocate a communication request object and associate it with the request 10handle (the argument request). The request can be used later to query the status of the 11communication or wait for its completion. 12A nonblocking send call indicates that the system may start copying data out of the 13send buffer. The sender should not modify any part of the send buffer after a nonblocking 14send operation is called, until the send completes. 15A nonblocking receive call indicates that the system may start writing data into the re-16ceive buffer. The receiver should not access any part of the receive buffer after a nonblocking 17receive operation is called, until the receive completes. 18 19Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10-2017.1.20. (End of advice to users.) 212223Communication Completion 3.7.3  $^{24}$ The functions MPI\_WAIT and MPI\_TEST are used to complete a nonblocking communica-25tion. The completion of a send operation indicates that the sender is now free to update the 26locations in the send buffer (the send operation itself leaves the content of the send buffer 27unchanged). It does not indicate that the message has been received, rather, it may have 28 been buffered by the communication subsystem. However, if a synchronous mode send was 29 used, the completion of the send operation indicates that a matching receive was initiated, 30 and that the message will eventually be received by this matching receive.  $^{31}$ The completion of a receive operation indicates that the receive buffer contains the 32 received message, the receiver is now free to access it, and that the status object is set. It 33 does not indicate that the matching send operation has completed (but indicates, of course, 34that the send was initiated). 35 We shall use the following terminology: A *null* handle is a handle with value 36 MPI\_REQUEST\_NULL. A persistent request and the handle to it are *inactive* if the request 37 is not associated with any ongoing communication (see Section 3.9 on page 73). A handle 38 is *active* if it is neither null nor inactive. An *empty* status is a status which is set to 39 return tag = MPI\_ANY\_TAG, source = MPI\_ANY\_SOURCE, error = MPI\_SUCCESS, and is 40 also internally configured so that calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and 41  $MPI_GET_ELEMENTS_X$  return count = 0 and  $MPI_TEST_CANCELLED$  returns false. We 42set a status variable to empty when the value returned by it is not significant. Status is set 43 in this way so as to prevent errors due to accesses of stale information. 44The fields in a status object returned by a call to MPI\_WAIT, MPI\_TEST, or any

<sup>45</sup> The fields in a status object returned by a call to MPI\_WAIT, MPI\_TEST, or any <sup>46</sup> of the other derived functions (MPI\_{TEST|WAIT}{ALL|SOME|ANY}), where the request <sup>47</sup> corresponds to a send call, are undefined, with two exceptions: The error status field will

#### 3.7. NONBLOCKING COMMUNICATION

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)
INOUT request mequest (handle)
OUT status status object (Status)
int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status)
MPI\_Wait(request, status, ierror) BIND(C)
TYPE(MPI\_Request), INTENT(INOUT) :: request
TYPE(MPI\_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI\_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR

A call to MPI\_WAIT returns when the operation identified by request is complete. If the request is an active persistent request, it is marked inactive. Any other type of request is and the request handle is set to MPI\_REQUEST\_NULL. MPI\_WAIT is a non-local operation.

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 multithreaded 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.*)  $^{24}$ 

 $^{31}$ 

```
1
     MPI_TEST(request, flag, status)
2
       INOUT
                 request
                                              communication request (handle)
3
       OUT
                 flag
                                              true if operation completed (logical)
4
5
       OUT
                                              status object (Status)
                 status
6
\overline{7}
     int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
8
     MPI_Test(request, flag, status, ierror) BIND(C)
9
          TYPE(MPI_Request), INTENT(INOUT) :: request
10
          LOGICAL, INTENT(OUT) :: flag
11
          TYPE(MPI_Status) :: status
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
15
          LOGICAL FLAG
16
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
17
          A call to MPI_TEST returns flag = true if the operation identified by request is complete.
18
     In such a case, the status object is set to contain information on the completed operation.
19
     If the request is an active persistent request, it is marked as inactive. Any other type of
20
     request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns
21
     flag = false if the operation identified by request is not complete. In this case, the value of
22
     the status object is undefined. MPI_TEST is a local operation.
23
          The return status object for a receive operation carries information that can be accessed
24
     as described in Section 3.2.5. The status object for a send operation carries information
25
     that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).
26
          One is allowed to call MPI_TEST with a null or inactive request argument. In such a
27
     case the operation returns with flag = true and empty status.
28
          The functions MPI_WAIT and MPI_TEST can be used to complete both sends and
29
     receives.
30
^{31}
                               The use of the nonblocking MPI_TEST call allows the user to
           Advice to users.
32
           schedule alternative activities within a single thread of execution. An event-driven
33
           thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to
34
           users.)
35
36
37
     Example 3.11
                       Simple usage of nonblocking operations and MPI_WAIT.
38
39
     CALL MPI_COMM_RANK(comm, rank, ierr)
40
     IF (rank.EQ.0) THEN
41
          CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
42
          **** do some computation to mask latency ****
43
          CALL MPI_WAIT(request, status, ierr)
44
     ELSE IF (rank.EQ.1) THEN
45
          CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
46
          **** do some computation to mask latency ****
47
          CALL MPI_WAIT(request, status, ierr)
48
     END IF
```

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

	3
MPI_REQUEST_FREE(request)	4
	5 6
INOUT request communication request (handle)	7
	8
<pre>int MPI_Request_free(MPI_Request *request)</pre>	9
MPI_Request_free(request, ierror) BIND(C)	10
TYPE(MPI_Request), INTENT(INOUT) :: request	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_REQUEST_FREE(REQUEST, IERROR)	13
INTEGER REQUEST, IERROR	14
	15
Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An	16
ongoing communication that is associated with the request will be allowed to complete. The	17
request will be deallocated only after its completion.	18
Rationale. The MPI_REQUEST_FREE mechanism is provided for reasons of perfor-	19
mance and convenience on the sending side. ( <i>End of rationale.</i> )	20
mance and convenience on the sending side. ( <i>Dia of Factoriae.</i> )	21
Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not	22 23
possible to check for the successful completion of the associated communication with	23
calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the	25
communication, an error code cannot be returned to the user — such an error must	26
be treated as fatal. An active receive request should never be freed as the receiver	27
will have no way to verify that the receive has completed and the receive buffer can	28
be reused. (End of advice to users.)	29
	30
<b>Example 3.12</b> An example using MPI_REQUEST_FREE.	31
	32
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)	33
IF (rank.EQ.0) THEN	34
DO i=1, n	35
CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	36
CALL MPI_REQUEST_FREE(req, ierr)	37
CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	38
CALL MPI_WAIT(req, status, ierr)	39
END DO	40 41
ELSE IF (rank.EQ.1) THEN	41
CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	43
CALL MPI_WAIT(req, status, ierr)	44
DO I=1, n-1 CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	45
CALL MPI_ISEND(OULVAI, I, MFI_REAL, 0, 0, MFI_COMM_WORLD, Teq, TeTT) CALL MPI_REQUEST_FREE(req, ierr)	46
CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	47
CALL MPI_WAIT(req, status, ierr)	48

1

 $\frac{2}{3}$ 

```
1
          END DO
\mathbf{2}
          CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
3
          CALL MPI_WAIT(req, status, ierr)
4
     END IF
5
6
            Semantics of Nonblocking Communications
     3.7.4
\overline{7}
     The semantics of nonblocking communication is defined by suitably extending the definitions
8
     in Section 3.5.
9
10
     Order Nonblocking communication operations are ordered according to the execution order
11
     of the calls that initiate the communication. The non-overtaking requirement of Section 3.5
12
     is extended to nonblocking communication, with this definition of order being used.
13
14
                       Message ordering for nonblocking operations.
     Example 3.13
15
16
     CALL MPI_COMM_RANK(comm, rank, ierr)
17
     IF (RANK.EQ.O) THEN
18
            CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
19
            CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
20
     ELSE IF (rank.EQ.1) THEN
21
            CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
22
            CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
23
     END IF
^{24}
     CALL MPI_WAIT(r1, status, ierr)
25
     CALL MPI_WAIT(r2, status, ierr)
26
27
     The first send of process zero will match the first receive of process one, even if both messages
28
     are sent before process one executes either receive.
29
30
     Progress A call to MPI_WAIT that completes a receive will eventually terminate and return
^{31}
     if a matching send has been started, unless the send is satisfied by another receive. In
32
     particular, if the matching send is nonblocking, then the receive should complete even if no
33
     call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that
34
     completes a send will eventually return if a matching receive has been started, unless the
35
     receive is satisfied by another send, and even if no call is executed to complete the receive.
36
                        An illustration of progress semantics.
37
     Example 3.14
38
     CALL MPI_COMM_RANK(comm, rank, ierr)
39
     IF (RANK.EQ.O) THEN
40
            CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
41
            CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
42
     ELSE IF (rank.EQ.1) THEN
43
            CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
44
            CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
45
            CALL MPI_WAIT(r, status, ierr)
46
     END IF
47
48
```

This code should not deadlock in a correct MPI implementation. The first synchronous send of process zero must complete after process one posts the matching (nonblocking) receive even if process one has not yet reached the completing wait call. Thus, process zero will continue and execute the second send, allowing process one to complete execution.

If an MPI\_TEST that completes a receive is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI\_TEST that completes a send is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return flag = true, unless the receive is satisfied by another send.

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI\_WAITANY or MPI\_TESTANY can be used to wait for the completion of one out of several operations. A call to MPI\_WAITALL or MPI\_TESTALL can be used to wait for all pending operations in a list. A call to MPI\_WAITSOME or MPI\_TESTSOME can be used to complete all enabled operations in a list.

MPI\_WAITANY (count, array\_of\_requests, index, status)

INTEGER, OPTIONAL, INTENT(OUT) ::

	Wiri _ wiri rite (count, anay_or_requests, index, status)		
IN	count	list length (non-negative integer)	22
INOUT	array_of_requests	array of requests (array of handles)	23
OUT	index	index of handle for operation that completed (integer)	24 25
OUT	status	status object (Status)	25 26
<pre>int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,</pre>			28
MPI_Status *status)			29
MDT Voite	nu (count orrow of rows)	ta inder status ierren) DIND(C)	30
	· · · ·	sts, index, status, ierror) BIND(C)	31
	ER, INTENT(IN) :: count		32
	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)		
	INTEGER, INTENT(OUT) :: index		
TYPE(	MPI_Status) :: status		35

ierror

MPI\_WAITANY(COUNT, ARRAY\_OF\_REQUESTS, INDEX, STATUS, IERROR)
INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), INDEX, STATUS(MPI\_STATUS\_SIZE),
IERROR

Blocks until one of the operations associated with the active requests in the array has completed. If more than 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 operation. (The array is indexed from zero in C, and from one in Fortran.) If the request 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

```
1
      immediately with index = MPI_UNDEFINED, and an empty status.
\mathbf{2}
          The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same
3
      effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value
4
      returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an
\mathbf{5}
      array containing one active entry is equivalent to MPI_WAIT.
6
7
      MPI_TESTANY(count, array_of_requests, index, flag, status)
8
9
       IN
                  count
                                               list length (non-negative integer)
10
       INOUT
                  array_of_requests
                                               array of requests (array of handles)
11
       OUT
                 index
                                               index of operation that completed, or
12
                                               MPI_UNDEFINED if none completed (integer)
13
14
        OUT
                 flag
                                               true if one of the operations is complete (logical)
15
       OUT
                 status
                                               status object (Status)
16
17
      int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,
18
                     int *flag, MPI_Status *status)
19
20
     MPI_Testany(count, array_of_requests, index, flag, status, ierror) BIND(C)
21
          INTEGER, INTENT(IN) :: count
22
          TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
23
          INTEGER, INTENT(OUT) ::
                                        index
24
          LOGICAL, INTENT(OUT) ::
                                        flag
25
          TYPE(MPI_Status) :: status
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
28
          LOGICAL FLAG
29
          INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
30
          IERROR
31
32
          Tests for completion of either one or none of the operations associated with active
33
      handles. In the former case, it returns flag = true, returns in index the index of this request
34
      in the array, and returns in status the status of that operation. If the request is an active
35
      persistent request, it is marked as inactive. Any other type of request is deallocated and
36
      the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from
37
      one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns
38
      a value of MPI_UNDEFINED in index and status is undefined.
39
          The array may contain null or inactive handles. If the array contains no active handles
40
      then the call returns immediately with flag = true, index = MPI_UNDEFINED, and an empty
^{41}
      status.
42
          If the array of requests contains active handles then the execution of
43
      MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution
^{44}
      of MPI_TEST( &array_of_requests[i], flag, status), for i=0, 1, ..., count-1, in some arbitrary
45
      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

array containing one active entry is equivalent to MPI\_TEST.

46

47

MPI_WAITALL( count, array_of_requests, array_of_statuses)			1
IN	count	lists length (non-negative integer)	2
		,	3
INOUT	array_of_requests	array of requests (array of handles)	4
OUT	array_of_statuses	array of status objects (array of Status)	5
			6
int MPI W	aitall(int count, MPI Re	<pre>quest array_of_requests[],</pre>	7
	MPI_Status array_of_		8
			9
<pre>MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) BIND(C)</pre>			10
			11
	-	UT) :: array_of_requests(count)	12
	MPI_Status) :: array_of		13
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	14
ΜΡΤ ΨΑΤΤΑ	LL COUNT, ARRAY OF REQUE	STS, ARRAY_OF_STATUSES, IERROR)	15
	ER COUNT, ARRAY_OF_REQUE		16
	ER ARRAY_OF_STATUSES(MPI)		17
			18
Blocks	Blocks until all communication operations associated with active handles in the list		

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 i-th entry in array\_of\_statuses is set to the return status of the i-th 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. 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 arbitrary 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 desirable to return specific information on each communication. The function MPI\_WAITALL will return in such case the error code MPI\_ERR\_IN\_STATUS and will set the error field of each status to a specific error code. This code will be MPI\_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI\_ERR\_PENDING if it has neither failed nor completed. The function MPI\_WAITALL will return MPI\_SUCCESS if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

*Rationale.* 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. It needs to check each individual status only when an error occurred. (End of rationale.) 20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38 39

40

41

```
1
      MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)
\mathbf{2}
       IN
                 count
                                               lists length (non-negative integer)
3
       INOUT
                 array_of_requests
                                               array of requests (array of handles)
4
5
       OUT
                 flag
                                               (logical)
6
        OUT
                 array_of_statuses
                                               array of status objects (array of Status)
7
8
      int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
9
                     MPI_Status array_of_statuses[])
10
11
     MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
12
                     BIND(C)
13
          INTEGER, INTENT(IN) :: count
14
          TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
15
          LOGICAL, INTENT(OUT) :: flag
16
          TYPE(MPI_Status) :: array_of_statuses(*)
17
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
19
          LOGICAL FLAG
20
          INTEGER COUNT, ARRAY_OF_REQUESTS(*),
21
          ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
22
23
          Returns flag = true if all communications associated with active handles in the array
^{24}
      have completed (this includes the case where no handle in the list is active). In this case, each
25
      status entry that corresponds to an active request is set to the status of the corresponding
26
      operation. Active persistent requests are marked inactive. Requests of any other type are
27
      deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL.
28
      Each status entry that corresponds to a null or inactive handle is set to empty.
29
          Otherwise, flag = false is returned, no request is modified and the values of the status
30
     entries are undefined. This is a local operation.
^{31}
          Errors that occurred during the execution of MPI_TESTALL are handled in the same
32
     manner as errors in MPI_WAITALL.
33
34
      MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)
35
36
37
       IN
                 incount
                                               length of array_of_requests (non-negative integer)
38
       INOUT
                 array_of_requests
                                               array of requests (array of handles)
39
        OUT
                 outcount
                                               number of completed requests (integer)
40
41
        OUT
                 array_of_indices
                                               array of indices of operations that completed (array of
42
                                               integers)
43
       OUT
                 array_of_statuses
                                               array of status objects for operations that completed
44
                                               (array of Status)
45
46
      int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
47
                     int *outcount, int array_of_indices[],
48
```

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 have completed. Returns in the first outcount locations of the array array\_of\_indices the indices of these operations (index within the array array\_of\_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array\_of\_status the status for these completed operations. Completed active persistent requests are marked as inactive. Any other type or request that completed is deallocated, and the associated handle is set to MPI\_REQUEST\_NULL.

If the list contains no active handles, then the call returns immediately with  $outcount = MPI_UNDEFINED$ .

When one or more of the communications completed by MPI\_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array\_of\_indices and array\_of\_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI\_ERR\_IN\_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI\_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI\_TESTSOME(incount, array\_of\_requests, outcount, array\_of\_indices, array\_of\_statuses)

				37
	IN	incount	length of array_of_requests (non-negative integer) $% \left( \left( {{{\left( {{{\left( {{\left( {{\left( {{\left( {{{\left( {{{\left( {{\left( {{\left( {{\left( {{\left( {{{\left( {{{\left( {{{\left( {{{\left( {{{\left( {{{\left( {{{}}}}} \right)}}}} \right. \\ ({{\left( {{\left( {{{\left( {{{}}}} \right)}}} \right)}} \right)}} \right)} \right)} \right)} \right)} \right)}$	38
	INOUT	array_of_requests	array of requests (array of handles)	39
	OUT	outcount	number of completed requests (integer)	40
	0.U.T	с.:		41
	OUT	array_of_indices	array of indices of operations that completed (array of	42
			integers)	43
	OUT	array_of_statuses	array of status objects for operations that completed	44
			(array of Status)	45
				46
÷	∽+ МDТ Т	octoomo(int incount MDT	Poqueat array of requests[]	47

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

1	MPI_Status array_of_statuses[])
2	MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
3	array_of_statuses, ierror) BIND(C)
4	INTEGER, INTENT(IN) :: incount
5 6	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
7	INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
8	TYPE(MPI_Status) :: array_of_statuses(*)
9	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10	MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
11	ARRAY_OF_STATUSES, IERROR)
12	INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
13	ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
14	Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has
15	completed it returns outcount = 0. If there is no active handle in the list it returns outcount
16 17	$=$ MPI_UNDEFINED.
17	MPI_TESTSOME is a local operation, which returns immediately, whereas
19	MPI_WAITSOME will block until a communication completes, if it was passed a list that
20	contains at least one active handle. Both calls fulfill a <i>fairness</i> requirement: If a request
21	for a receive repeatedly appears in a list of requests passed to $MPI_WAITSOME$ or
22	MPI_TESTSOME, and a matching send has been posted, then the receive will eventually
23	succeed, unless the send is satisfied by another receive; and similarly for send requests.
24	Errors that occur during the execution of MPI_TESTSOME are handled as for
25	MPI_WAITSOME.
26	Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use
27	of MPI_TESTANY. The former returns information on all completed communications,
28 29	with the latter, a new call is required for each communication that completes.
30	A server with multiple clients can use MPI_WAITSOME so as not to starve any client.
31	Clients send messages to the server with service requests. The server calls
32	$MPI\_WAITSOME$ with one receive request for each client, and then handles all receives
33	that completed. If a call to $MPI\_WAITANY$ is used instead, then one client could starve
34	while requests from another client always sneak in first. (End of advice to users.)
35	Advise to implementance MDL TESTSOME should complete as many pending com
36	Advice to implementors. MPI_TESTSOME should complete as many pending com- munications as possible. ( <i>End of advice to implementors.</i> )
37	
38 39	
40	<b>Example 3.15</b> Client-server code (starvation can occur).
41	
42	CALL MPI_COMM_SIZE(comm, size, ierr)
43	CALL MPI_COMM_RANK(comm, rank, ierr)
44	IF(rank .GT. 0) THEN ! client code
45	DO WHILE(.TRUE.)
46	CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
47	CALL MPI_WAIT(request, status, ierr)
48	END DO

```
! rank=0 -- server code
                                                                                      1
ELSE
                                                                                      \mathbf{2}
       DO i=1, size-1
                                                                                      3
           CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                     comm, request_list(i), ierr)
                                                                                      4
       END DO
                                                                                      5
                                                                                      6
       DO WHILE(.TRUE.)
                                                                                      7
           CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
                                                                                      8
           CALL DO_SERVICE(a(1, index)) ! handle one message
           CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
                                                                                      9
                                                                                      10
                      comm, request_list(index), ierr)
       END DO
                                                                                      11
END IF
                                                                                      12
                                                                                      13
                                                                                      14
Example 3.16
                 Same code, using MPI_WAITSOME.
                                                                                      15
                                                                                      16
                                                                                      17
CALL MPI_COMM_SIZE(comm, size, ierr)
                                                                                      18
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                      19
IF(rank .GT. 0) THEN
                               ! client code
                                                                                      20
    DO WHILE(.TRUE.)
                                                                                      21
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
                                                                                      22
        CALL MPI_WAIT(request, status, ierr)
                                                                                      23
    END DO
                                                                                      ^{24}
ELSE
              ! rank=0 -- server code
                                                                                      25
    DO i=1, size-1
                                                                                      26
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                                                                                      27
                        comm, request_list(i), ierr)
                                                                                      28
    END DO
                                                                                      29
    DO WHILE(.TRUE.)
                                                                                      30
       CALL MPI_WAITSOME(size, request_list, numdone,
                                                                                      31
                          indices, statuses, ierr)
                                                                                      32
       DO i=1, numdone
                                                                                      33
           CALL DO_SERVICE(a(1, indices(i)))
                                                                                      34
           CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                                                                                      35
                         comm, request_list(indices(i)), ierr)
                                                                                      36
       END DO
                                                                                      37
    END DO
                                                                                      38
END IF
                                                                                      39
                                                                                      40
```

#### 3.7.6 Non-destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information. 41

42

43

44

MPI\_REQUEST\_GET\_STATUS( request, flag, status )

```
2
       IN
                 request
                                              request (handle)
3
       OUT
                 flag
                                              boolean flag, same as from MPI_TEST (logical)
4
       OUT
                                              status object if flag is true (Status)
                 status
5
6
\overline{7}
     int MPI_Request_get_status(MPI_Request request, int *flag,
8
                     MPI_Status *status)
9
     MPI_Request_get_status(request, flag, status, ierror) BIND(C)
10
          TYPE(MPI_Request), INTENT(IN) :: request
11
          LOGICAL, INTENT(OUT) :: flag
12
          TYPE(MPI_Status) :: status
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
14
15
     MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
16
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
17
          LOGICAL FLAG
18
          Sets flag=true if the operation is complete, and, if so, returns in status the request
19
     status. However, unlike test or wait, it does not deallocate or inactivate the request; a
20
     subsequent call to test, wait or free should be executed with that request. It sets flag=false
21
     if the operation is not complete.
22
          One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request
23
```

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.

25 26

27

## 3.8 Probe and Cancel

The MPI\_PROBE, 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 cancelled. 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

38 39 40

41

36 37

MPI\_IPROBE(source, tag, comm, flag, status)

42	IN	source	rank of source or $MPI_ANY_SOURCE$ (integer)
43	IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)
44	IN	comm	communicator (handle)
45 46	OUT	flag	(logical)
47	OUT	status	status object (Status)
48			

<pre>int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)</pre>
<pre>MPI_Iprobe(source, tag, comm, flag, status, ierror) BIND(C) INTEGER, INTENT(IN) :: source, tag</pre>
TYPE(MPI_Comm), INTENT(IN) :: comm LOGICAL, INTENT(OUT) :: flag TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI\_IPROBE(source, tag, comm, flag, 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(). 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 subsequently 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 returned in status by MPI\_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multithreaded, it is the user's responsibility to ensure that the last condition holds.

The source argument of MPI\_PROBE can be MPI\_ANY\_SOURCE, and the tag argument can be MPI\_ANY\_TAG, so that one can probe for messages from an arbitrary source and/or 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.

A probe with MPI\_PROC\_NULL as source returns flag = true, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0; see Section 3.11 on page 80.

MPI\_PROBE(source, tag, comm, status)

IN	source	rank of source or $MPI_ANY_SOURCE$ (integer)	40
IN	tag	message tag or $MPI_ANY_TAG$ (integer)	41
IN	comm	communicator (handle)	42
τυο	status	status object (Status)	$43 \\ 44$
			45

int MPI\_Probe(int source, int tag, MPI\_Comm comm, MPI\_Status \*status) 46
MPI\_Probe(source, tag, comm, status, ierror) BIND(C)
47
48

```
1
         INTEGER, INTENT(IN) :: source, tag
\mathbf{2}
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         TYPE(MPI_Status) :: status
4
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
5
     MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
6
          INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
7
8
         MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns
9
     only after a matching message has been found.
10
         The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress:
^{11}
     if a call to MPI_PROBE has been issued by a process, and a send that matches the probe
12
     has been initiated by some process, then the call to MPI_PROBE will return, unless the
13
     message is received by another concurrent receive operation (that is executed by another
14
     thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a
15
     matching message has been issued, then the call to MPI_IPROBE will eventually return flag
16
     = true unless the message is received by another concurrent receive operation or matched
17
     by a concurrent matched probe.
18
19
     Example 3.17
         Use blocking probe to wait for an incoming message.
20
21
             CALL MPI_COMM_RANK(comm, rank, ierr)
22
             IF (rank.EQ.0) THEN
23
                 CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
24
             ELSE IF (rank.EQ.1) THEN
25
                 CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
26
             ELSE IF (rank.EQ.2) THEN
27
                 DO i=1, 2
28
                     CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
29
                                     comm, status, ierr)
30
                     IF (status(MPI_SOURCE) .EQ. 0) THEN
31
     100
                         CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
32
                     ELSE
33
                         CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
     200
34
                     END IF
35
                 END DO
36
             END IF
37
38
     Each message is received with the right type.
39
40
     Example 3.18
                      A similar program to the previous example, but now it has a problem.
41
42
             CALL MPI_COMM_RANK(comm, rank, ierr)
43
             IF (rank.EQ.0) THEN
44
                  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
45
             ELSE IF (rank.EQ.1) THEN
46
                  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
47
             ELSE IF (rank.EQ.2) THEN
48
                 DO i=1, 2
```

In Example 3.18, the two receive calls in statements labeled 100 and 200 in Example 3.17 slightly modified, using MPI\_ANY\_SOURCE as the source argument. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI\_PROBE.

Advice to users. In a multithreaded MPI program, MPI\_PROBE and 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 that found by the probe since another thread could concurrently receive that original message [29]. MPI\_MPROBE and MPI\_IMPROBE solve this problem by matching the incoming message so that it may only be received with MPI\_MRECV or MPI\_IMRECV on the corresponding message handle. (*End of advice to users.*)

Advice to implementors. A call to MPI\_PROBE(source, tag, comm, status) will match 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, tag t and communicator c. If the tag argument in the probe call has value MPI\_ANY\_TAG then the message probed will be the earliest pending message from source s with communicator c and any tag; in any case, the message probed will be the earliest pending message from source s with tag t and communicator c (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source s with tag t and communicator 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 the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors.*)

#### 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 infor mation returned by the probe. In particular, the user may allocate memory for the receive
 buffer, according to the length of the probed message.

- 4
- 5 6

 $\overline{7}$ 

8

9

10 11

12

13

14 15 16

17

18

19

20

21

22

23

 $^{24}$ 

25

```
MPI_IMPROBE(source, tag, comm, flag, message, status)
 IN
          source
                                     rank of source or MPI_ANY_SOURCE (integer)
 IN
                                     message tag or MPI_ANY_TAG (integer)
          tag
 IN
                                     communicator (handle)
          comm
 OUT
          flag
                                     flag (logical)
 OUT
           message
                                     returned message (handle)
 OUT
                                     status object (Status)
          status
int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
              MPI_Message *message, MPI_Status *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

<sup>29</sup> MPI\_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is <sup>31</sup> a message that can be received and that matches the pattern specified by the arguments <sup>32</sup> source, tag, and comm. The call matches the same message that would have been received <sup>33</sup> by a call to MPI\_RECV(..., source, tag, comm, status) executed at the same point in the <sup>34</sup> program and returns in status the same value that would have been returned by MPI\_RECV. <sup>35</sup> In addition, it returns in message a handle to the matched message. Otherwise, the call <sup>36</sup> 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. Each message returned by MPI\_IMPROBE must be received with either MPI\_MRECV or MPI\_IMRECV.

The source argument of MPI\_IMPROBE can be MPI\_ANY\_SOURCE, and the tag argument can be MPI\_ANY\_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

A synchronous send operation that is matched with MPI\_IMPROBE or MPI\_MPROBE will complete successfully only if both a matching receive is posted with MPI\_MRECV or 8

synchronou There which has MPI_MESS, A ma MPI_MESS, = MPI_AN or MPI_IM Ratio MPI_	is send. is a special predefined messa MPI_PROC_NULL as its source AGE_NULL is the value used for tching probe with MPI_PROC AGE_NO_PROC, and the statu IY_TAG, and count = 0; see Sec RECV with MPI_MESSAGE_NO ponale. MPI_MESSAGE_NO	n has started to receive the message sent by the ge: MPI_MESSAGE_NO_PROC, which is a message re process. The predefined constant r invalid message handles. C_NULL as source returns flag = true, message = as object returns source = MPI_PROC_NULL, tag pection 3.11. It is not necessary to call MPI_MRECV NO_PROC, but it is not erroneous to do so. _PROC was chosen instead of void possible confusion as another null handle con-	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
MPI_MPR	OBE(source, tag, comm, messa	ge, status)	16
IN	source	rank of source or MPI_ANY_SOURCE (integer)	17 18
IN	tag	message tag or MPI_ANY_TAG (integer)	19
IN	comm	communicator (handle)	20
			21
OUT	message	returned message (handle)	22
OUT	status	status object (Status)	23 24
<pre>int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,</pre>			25 26 27 28 29 30
	MPI_Message), INTENT(OUT) MPI_Status) :: status	:: message	31 32
	ER, OPTIONAL, INTENT(OUT)	:: ierror	33
			34
	E (SOURCE, TAG, COMM, MESS FER SOURCE TAG, COMM, MESS	SAGE, STATUS, IERROR) SSAGE, STATUS(MPI_STATUS_SIZE), IERROR	35
			36
	MPROBE behaves like MPI_IM a matching message has been	PROBE except that it is a blocking call that returns found.	37 38
	-	<b>BE</b> and <b>MPI_IMPROBE</b> needs to guarantee progress	39
in the sam	e way as in the case of $MPI_P$	ROBE and MPI_IPROBE.	40
			41
3.8.3 Ma	tched Receives		42 43
The functi	ons $MPI_MRECV$ and $MPI_IM$	IRECV receive messages that have been previously	43 44
matched b	y a matching probe (Section 3	3.8.2).	45
			46
			47
48			

1 MPI\_MRECV(buf, count, datatype, message, status) 2 OUT buf initial address of receive buffer (choice) 3 IN count number of elements in receive buffer (non-negative in-4 teger) 56 IN datatype of each receive buffer element (handle) datatype 7 INOUT message message (handle) 8 OUT status status object (Status) 9 10 11int MPI\_Mrecv(void\* buf, int count, MPI\_Datatype datatype, MPI\_Message \*message, MPI\_Status \*status) 1213MPI\_Mrecv(buf, count, datatype, message, status, ierror) BIND(C) 14TYPE(\*), DIMENSION(..) :: buf 15INTEGER, INTENT(IN) :: count 16TYPE(MPI\_Datatype), INTENT(IN) :: datatype 17 TYPE(MPI\_Message), INTENT(INOUT) :: message 18 TYPE(MPI\_Status) :: status 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 2122 <type> BUF(\*) 23INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 24This call receives a message matched by a matching probe operation (Section 3.8.2). 25The receive buffer consists of the storage containing **count** consecutive elements of the 26type specified by datatype, starting at address buf. The length of the received message must 27be less than or equal to the length of the receive buffer. An overflow error occurs if all 28incoming data does not fit, without truncation, into the receive buffer. 29 If the message is shorter than the receive buffer, then only those locations corresponding 30 to the (shorter) message are modified.  $^{31}$ On return from this function, the message handle is set to MPI\_MESSAGE\_NULL. All 32 errors that occur during the execution of this operation are handled according to the error 33 handler set for the communicator used in the matching probe call that produced the message 34handle. 35 If MPI\_MRECV is called with MPI\_MESSAGE\_NO\_PROC as the message argument, the 36 call returns immediately with the status object set to source = MPI\_PROC\_NULL, tag =37 MPI\_ANY\_TAG, and count = 0, as if a receive from MPI\_PROC\_NULL was issued (see Sec-38 tion 3.11). A call to MPI\_MRECV with MPI\_MESSAGE\_NULL is erroneous. 39 40 41 4243 44454647 48

MPI_IMRE	CV(buf, count, datatype, mess	age, request)	1
OUT	buf	initial address of receive buffer (choice)	2
IN	count	number of elements in receive buffer (non-negative in-	3
	count	teger)	4 5
IN	datatype	datatype of each receive buffer element (handle)	6
INOUT	message	message (handle)	7
	-		8
OUT	request	communication request (handle)	9
int MDT I	magan (maidt buf int agu	nt MDT Detetune detetune	10
int MPI_I		nt, MPI_Datatype datatype, , MPI_Request *request)	11
			12 13
		message, request, ierror) BIND(C)	14
	(*), DIMENSION(), ASYNC	HRONOUS :: buf	15
	ER, INTENT(IN) :: count MPI_Datatype), INTENT(IN	) :: datatype	16
	(MPI_Message), INTENT(INO	V-1	17
	(MPI_Request), INTENT(OUT)	0	18
	ER, OPTIONAL, INTENT(OUT	-	19
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)			20
	<pre>&gt; BUF(*)</pre>	HESSAGE, HEQUEST, TERRORY	21 22
• 1	ER COUNT, DATATYPE, MESS	AGE, REQUEST, IERROR	23
MDLI	MPECV is the nephledring	variant of MPI_MRECV and starts a nonblocking	24
	9	on semantics are similar to MPI_IRECV as described	25
	<u> </u>	function, the message handle is set to	26
MPI_MESS			27
If MP	$I\_IMRECV$ is called with MPI_	MESSAGE_NO_PROC as the message argument, the	28 29
	· · ·	object which, when completed, will yield a status	29 30
-		L, tag = MPI_ANY_TAG, and count = 0, as if a	31
		d (see Section 3.11). A call to $MPI_IMRECV$ with	32
MPI_MESS	AGE_NULL is erroneous.		33
Advi	ce to implementors. If recep	ption of a matched message is started with	34
	· -	cancel the returned request with MPI_CANCEL. If	35
	,	ed message must be found by a subsequent message	36
-		E, MPI_MPROBE, or MPI_IMPROBE), received by	37 38
		cancelled by the sender. See Section 3.8.4 for details tion of operations initiated with MPI_IMRECV may	39
	(End of advice to implemento	* · · · · · · · · · · · · · · · · · · ·	40
1011.			

3.8.4 Cancel

MPI\_CANCEL(request)

communication request (handle)

71

> 47 48

```
int MPI_Cancel(MPI_Request *request)
MPI_Cancel(request, ierror) BIND(C)
TYPE(MPI_Request), INTENT(IN) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_CANCEL(REQUEST, IERROR)
T INTEGER REQUEST, IERROR
```

A call to MPI\_CANCEL marks for cancellation a pending, nonblocking communication 9 operation (send or receive). The cancel call is local. It returns immediately, possibly before 10 the communication is actually cancelled. It is still necessary to call MPI\_REQUEST\_FREE. 11 MPI\_WAIT or MPI\_TEST (or any of the derived operations) with the cancelled request as 12argument after the call to MPI\_CANCEL. If a communication is marked for cancellation, 13 then a MPI\_WAIT call for that communication is guaranteed to return, irrespective of 14 the activities of other processes (i.e., MPI\_WAIT behaves as a local function); similarly if 15MPI\_TEST is repeatedly called in a busy wait loop for a cancelled communication, then 16MPI\_TEST will eventually be successful. 17

<sup>18</sup> MPI\_CANCEL can be used to cancel a communication that uses a persistent request (see <sup>19</sup> Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation <sup>20</sup> cancels the active communication, but not the request itself. After the call to MPI\_CANCEL <sup>21</sup> and the subsequent call to MPI\_WAIT or MPI\_TEST, the request becomes inactive and can <sup>22</sup> 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 25send is marked for cancellation, then it must be the case that either the send completes 26normally, in which case the message sent was received at the destination process, or that 27the send is successfully cancelled, in which case no part of the message was received at the 28destination. Then, any matching receive has to be satisfied by another send. If a receive is 29 marked for cancellation, then it must be the case that either the receive completes normally, 30 or that the receive is successfully cancelled, in which case no part of the receive buffer is  $^{31}$ altered. Then, any matching send has to be satisfied by another receive. 32

If the operation has been cancelled, 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)
```

43	IN	status	status object (Status)			
44	OUT	flag	(logical)			
45			(8)			
46	int MDT	Fost concolled(conc	+ MDT Ctatua watat	a int wflow)		
47	<pre>int MPI_Test_cancelled(const MPI_Status *status, int *)</pre>					
48	MPI_Test	_cancelled(status,	flag, ierror) BIND(	(C)		

8

33

34 35

36

37

38

39 40 41

TYPE(MPI_Status), INTENT(IN) :: status							
LOGICAL, INTENT(OUT) :: flag							
INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)							
LOGICAL FLAG							
INTEGER STATUS(MPI_STATUS_SIZE), IERROR							

Returns flag = true if the communication associated with the status object was cancelled 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 cancelled then one should call MPI\_TEST\_CANCELLED first, to check whether the operation was cancelled, before checking on the other fields of the return status.

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (End of advice to users.)

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 of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement

MPI\_CANCEL, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (*End of advice to implementors.*)

## 3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a *persistent* communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "halfchannel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the five following calls. These calls involve no communication.

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

1 MPI\_SEND\_INIT(buf, count, datatype, dest, tag, comm, request) 2 IN buf initial address of send buffer (choice) 3 IN count number of elements sent (non-negative integer) 4 5IN datatype type of each element (handle) 6 IN dest rank of destination (integer) 7 IN message tag (integer) tag 8 9 IN communicator (handle) comm 10 OUT request communication request (handle) 11 12int MPI\_Send\_init(const void\* buf, int count, MPI\_Datatype datatype, 13 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 1415MPI\_Send\_init(buf, count, datatype, dest, tag, comm, request, ierror) 16BIND(C) 17 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 18 INTEGER, INTENT(IN) :: count, dest, tag 19 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 20TYPE(MPI\_Comm), INTENT(IN) :: comm 21TYPE(MPI\_Request), INTENT(OUT) :: request 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23MPI\_SEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 24<type> BUF(\*) 25INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 2627Creates a persistent communication request for a standard mode send operation, and 28binds to it all the arguments of a send operation. 29 30 MPI\_BSEND\_INIT(buf, count, datatype, dest, tag, comm, request) 3132 IN buf initial address of send buffer (choice) 33 IN count number of elements sent (non-negative integer) 34 IN datatype type of each element (handle) 35 36 IN dest rank of destination (integer) 37 IN tag message tag (integer) 38 39 IN comm communicator (handle) 40OUT request communication request (handle) 41 42int MPI\_Bsend\_init(const void\* buf, int count, MPI\_Datatype datatype, 43 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 4445MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C) 4647 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 48 INTEGER, INTENT(IN) :: count, dest, tag

TYPE TYPE	(MPI_Datatype), INTENT(IN) (MPI_Comm), INTENT(IN) :: (MPI_Request), INTENT(OUT) GER, OPTIONAL, INTENT(OUT)	comm ) :: request	1 2 3 4		
<type INTE</type 	e> BUF(*) GER COUNT, DATATYPE, DEST	PE, DEST, TAG, COMM, REQUEST, IERROR) , TAG, COMM, REQUEST, IERROR	5 6 7 8 9		
Creat	es a persistent communication	a request for a buffered mode send.	9 10 11		
MPI_SSEN	ND_INIT(buf, count, datatype,	dest, tag, comm, request)	12		
IN	buf	initial address of send buffer (choice)	13		
IN	count	number of elements sent (non-negative integer)	14 15		
IN	datatype	type of each element (handle)	16		
IN	dest	rank of destination (integer)	17		
IN	tag	message tag (integer)	18 19		
IN	comm	communicator (handle)	20		
OUT	request	communication request (handle)	21 22		
<pre>int MPI_Ssend_init(const void* buf, int count, MPI_Datatype datatype,</pre>					
<pre>MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
<type< td=""><td colspan="5">MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type></td></type<>	MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>				
Creat	es a persistent communication	object for a synchronous mode send operation.	37 38 39 40 41 42 43 44 45		

```
1
     MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)
\mathbf{2}
       IN
                 buf
                                             initial address of send buffer (choice)
3
       IN
                 count
                                             number of elements sent (non-negative integer)
4
5
       IN
                 datatype
                                             type of each element (handle)
6
       IN
                 dest
                                             rank of destination (integer)
7
       IN
                                             message tag (integer)
                 tag
8
9
       IN
                                             communicator (handle)
                 comm
10
       OUT
                 request
                                             communication request (handle)
11
12
     int MPI_Rsend_init(const void* buf, int count, MPI_Datatype datatype,
13
                     int dest, int tag, MPI_Comm comm, MPI_Request *request)
14
15
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
16
                    BIND(C)
17
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
18
          INTEGER, INTENT(IN) :: count, dest, tag
19
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
          TYPE(MPI_Comm), INTENT(IN) :: comm
21
          TYPE(MPI_Request), INTENT(OUT) :: request
22
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
23
     MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
24
          <type> BUF(*)
25
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
26
27
          Creates a persistent communication object for a ready mode send operation.
28
29
     MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)
30
^{31}
       OUT
                 buf
                                             initial address of receive buffer (choice)
32
       IN
                 count
                                             number of elements received (non-negative integer)
33
34
       IN
                 datatype
                                             type of each element (handle)
35
       IN
                 source
                                             rank of source or MPI_ANY_SOURCE (integer)
36
       IN
                                             message tag or MPI_ANY_TAG (integer)
                 tag
37
       IN
                 comm
                                             communicator (handle)
38
39
       OUT
                 request
                                             communication request (handle)
40
41
     int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
42
                     int tag, MPI_Comm comm, MPI_Request *request)
43
     MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
44
45
                    BIND(C)
46
          TYPE(*), DIMENSION(...), ASYNCHRONOUS ::
                                                         buf
47
          INTEGER, INTENT(IN) :: count, source, tag
48
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

TYPE(MPI_Comm), INTENT(IN) ::	comm	1			
TYPE(MPI_Request), INTENT(OUT	) :: request	2			
INTEGER, OPTIONAL, INTENT(OUT	) :: ierror	3			
MPT RECV INIT (RILE COUNT DATATVP	E, SOURCE, TAG, COMM, REQUEST, IERROR)	4			
<pre><type> BUF(*)</type></pre>		5			
	CE, TAG, COMM, REQUEST, IERROR	6			
		7 8			
Creates a persistent communication request for a receive operation. The argument buf					
is marked as OUT because the user gives permission to write on the receive buffer by passing					
the argument to MPI_RECV_INIT.		10 11			
	t is inactive after it was created — no active com-	11			
munication is attached to the request.		12			
	) that uses a persistent request is initiated by the	14			
function MPI_START.		15			
		16			
MPI_START(request)		17			
INOUT request	communication request (handle)	18			
inour request	communication request (nandie)	19			
int MDI Ctant (MDI Dequest these	+ )	20			
<pre>int MPI_Start(MPI_Request *reques</pre>	τ)	21			
<pre>MPI_Start(request, ierror) BIND(C</pre>	)	22			
TYPE(MPI_Request), INTENT(INO	UT) :: request	23			
INTEGER, OPTIONAL, INTENT(OUT	) :: ierror	24			
MPI_START(REQUEST, IERROR)		25			
INTEGER REQUEST, IERROR		26			
		27			
<b>e</b> , , , ,	e returned by one of the previous five calls. The	28			
-	The request becomes active once the call is made.	29			
_	dy mode, then a matching receive should be posted	30 31			
and until the operation completes.	ation buffer should not be modified after the call,	31 32			
	ntics to the nonblocking communication operations	33			
	1 to MPI_START with a request created by	34			
,	on in the same manner as a call to MPI_ISEND; a	35			
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.				
		38			
		39			
MPI_STARTALL(count, array_of_request	s)	40			
IN count	list length (non-negative integer)	41			
INOUT array_of_requests	array of requests (array of handle)	42			
	and, or requests (array or numero)	43			
int MPI_Startall(int count, MPI_R	aguagt array of requests[])	44			
Int MFI_Startall(Int Count, MPI_A	ednese array or redneses[])	45			
<pre>MPI_Startall(count, array_of_requests, ierror) BIND(C)</pre>					
INTEGER, INTENT(IN) :: count		47			
TYPE(MPI_Request), INTENT(INO	UT) :: array_of_requests(count)	48			

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI\_STARTALL(COUNT, ARRAY\_OF\_REQUESTS, IERROR) 3 INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), IERROR 4 5Start all communications associated with requests in array\_of\_requests. A call to 6 MPI\_STARTALL(count, array\_of\_requests) has the same effect as calls to  $\overline{7}$ MPI\_START (&array\_of\_requests[i]), executed for i=0,..., count-1, in some arbitrary order. 8 A communication started with a call to MPI\_START or MPI\_STARTALL is completed 9 by a call to MPI\_WAIT, MPI\_TEST, or one of the derived functions described in Sec-10 tion 3.7.5. The request becomes inactive after successful completion of such call. The re-11quest is not deallocated and it can be activated anew by an MPI\_START or MPI\_STARTALL 12call. 13 A persistent request is deallocated by a call to MPI\_REQUEST\_FREE (Section 3.7.3). 14The call to MPI\_REQUEST\_FREE can occur at any point in the program after the per-15sistent request was created. However, the request will be deallocated only after it becomes 16inactive. Active receive requests should not be freed. Otherwise, it will not be possible 17to check that the receive has completed. It is preferable, in general, to free requests when 18 they are inactive. If this rule is followed, then the functions described in this section will be 19invoked in a sequence of the form, Create (Start Complete)\* Free where \* indicates 20zero or more repetitions. If the same communication object is used in several concurrent 21threads, it is the user's responsibility to coordinate calls so that the correct sequence is 22obeyed. 23A send operation initiated with MPI\_START can be matched with any receive operation  $^{24}$ and, likewise, a receive operation initiated with MPI\_START can receive messages generated 25by any send operation. 2627Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10-2817.1.20. (End of advice to users.) 2930  $^{31}$ 3.10 Send-Receive 32 33 The *send-receive* operations combine in one call the sending of a message to one destination 34and the receiving of another message, from another process. The two (source and destina-35 tion) are possibly the same. A send-receive operation is very useful for executing a shift 36 operation across a chain of processes. If blocking sends and receives are used for such a shift, 37 then one needs to order the sends and receives correctly (for example, even processes send, 38 then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that 39 may lead to deadlock. When a send-receive operation is used, the communication subsys-40 tem takes care of these issues. The send-receive operation can be used in conjunction with 41 the functions described in Chapter 7 in order to perform shifts on various logical topologies. 42Also, 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.

78

43

44

45

MPI_SENI	DRECV(sendbuf, sendcount, se source, recvtag, comm, s	endtype, dest, sendtag, recvbuf, recvcount, recvtype, tatus)	1 $2$
IN	sendbuf	initial address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	4 5 6
IN	sendtype	type of elements in send buffer (handle)	7
IN	dest	rank of destination (integer)	8
IN	sendtag	send tag (integer)	9 10
OUT	recvbuf	initial address of receive buffer (choice)	10
IN	recvcount	number of elements in receive buffer (non-negative in-teger)	12 13
IN	recvtype	type of elements in receive buffer (handle)	14
IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)	15 16
IN	recvtag	receive tag or $MPI_ANY_TAG$ (integer)	17
IN	comm	communicator (handle)	18
OUT	status	status object (Status)	19 20
int MP1_S	int dest, int sendta	<pre>buf, int sendcount, MPI_Datatype sendtype, g, void *recvbuf, int recvcount, be, int source, int recvtag, MPI_Comm comm,</pre>	22 23 24 25
TYPE TYPE INTE recvi	<pre>recvcount, recvtype, BIND(C) (*), DIMENSION(), INTEN (*), DIMENSION() :: r GER, INTENT(IN) :: sendc</pre>	ecvbuf ount, dest, sendtag, recvcount, source, ) :: sendtype, recvtype	26 27 28 29 30 31 32 33 34
TYPE	(MPI_Status) :: status GER, OPTIONAL, INTENT(OUT		35 36
<type INTEC</type 	RECVCOUNT, RECVTYPE, > SENDBUF(*), RECVBUF(*) GER SENDCOUNT, SENDTYPE,	SENDTYPE, DEST, SENDTAG, RECVBUF, SOURCE, RECVTAG, COMM, STATUS, IERROR) DEST, SENDTAG, RECVCOUNT, RECVTYPE, (MPI_STATUS_SIZE), IERROR	37 38 39 40 41 42
	8	ve operation. Both send and receive use the same ags. The send buffer and receive buffers must be	43 44

communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

The semantics of a send-receive operation is what would be obtained if the caller forked <sup>46</sup> two concurrent threads, one to execute the send, and one to execute the receive, followed <sup>47</sup> by a join of these two threads. <sup>48</sup>

```
1
     MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, sta-
\mathbf{2}
                     tus)
3
       INOUT
                 buf
                                              initial address of send and receive buffer (choice)
4
       IN
                                              number of elements in send and receive buffer (non-
                 count
5
                                              negative integer)
6
7
       IN
                 datatype
                                              type of elements in send and receive buffer (handle)
8
       IN
                 dest
                                              rank of destination (integer)
9
                                              send message tag (integer)
       IN
                 sendtag
10
11
       IN
                 source
                                              rank of source or MPI_ANY_SOURCE (integer)
12
       IN
                 recvtag
                                              receive message tag or MPI_ANY_TAG (integer)
13
       IN
                                              communicator (handle)
                 comm
14
15
       OUT
                                              status object (Status)
                 status
16
17
     int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
18
                     int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
19
                     MPI_Status *status)
20
     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
21
                     comm, status, ierror) BIND(C)
22
          TYPE(*), DIMENSION(..)
                                     :: buf
23
          INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          TYPE(MPI_Status) :: status
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
30
                     COMM, STATUS, IERROR)
31
          <type> BUF(*)
32
          INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
33
          STATUS(MPI_STATUS_SIZE), IERROR
34
          Execute a blocking send and receive. The same buffer is used both for the send and
35
     for the receive, so that the message sent is replaced by the message received.
36
37
           Advice to implementors. Additional intermediate buffering is needed for the "replace"
38
           variant. (End of advice to implementors.)
39
40
41
             Null Processes
     3.11
42
43
     In many instances, it is convenient to specify a "dummy" source or destination for commu-
```

In many instances, it is convenient to specify a "dummy" source or destination for commu nication. 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 probe or matching probe with source = MPI\_PROC\_NULL succeeds and returns as soon as possible, and 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\_NO\_PROC, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0.

## Chapter 4

## Datatypes

Basic datatypes were introduced in Section 3.2.2 on page 25 and in Section 3.3 on page 33. 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.

7 8 9

15

16

17

18 19

2021

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

38

39

40

41

42

43

44

4546

47

48

#### 4.1**Derived** Datatypes

Up to here, all point to point communications 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 shapes and sizes. It is not assumed that the MPI library is cognizant of 34 the objects declared in the host language. Thus, if one wants to transfer a structure, or an 35 array section, it will be necessary to provide in MPI a definition of a communication buffer 36 that mimics the definition of the structure or array section in question. These facilities can 37 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
- 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.

6

1

 $\mathbf{2}$ 

3

4

5

 $\overline{7}$ 8 9

10

1112

13

14

15

18

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

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

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

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

16Most datatype constructors have replication count or block length arguments. Allowed 17values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent. 19

We can use a handle to a general datatype as an argument in a send or receive operation, 20instead of a basic datatype argument. The operation MPI\_SEND(buf, 1, datatype,...) will 21use the send buffer defined by the base address buf and the general datatype associated 22with datatype; it will generate a message with the type signature determined by the datatype 23argument. MPI\_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base  $^{24}$ address **buf** and the general datatype associated with **datatype**. 25

General datatypes can be used in all send and receive operations. We discuss, in 26Section 4.1.11 on page 111, the case where the second argument count has value > 1. 27

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, 28and are predefined. Thus, MPI\_INT is a predefined handle to a datatype with type map 29 $\{(int, 0)\}$ , with one entry of type int and displacement zero. The other basic datatypes 30 are similar.  $^{31}$ 

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), \dots, (type_{n-1}, disp_{n-1})\},\$$

`

then

38 39 40

41 42

32

33

3435

36

37

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

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

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

43If  $type_i$  requires alignment to a byte address that is a multiple of  $k_j$ , then  $\epsilon$  is the least 44non-negative increment needed to round extent(Typemap) to the next multiple of  $\max_i k_i$ . 45In Fortran, it is implementation dependent whether the MPI implementation computes 46the alignments  $k_i$  according to the alignments used by the compiler in common blocks, 47SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE 48nor BIND(C). The complete definition of *extent* is given in Section 4.1.6 on page 103.

Let

**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 103 and in Section 17.1.15 on page 633. (End of rationale.)

#### 4.1.1 Type Constructors with Explicit Addresses

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 are used in 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.

#### 4.1.2 Datatype Constructors

**Contiguous** The simplest datatype constructor is MPI\_TYPE\_CONTIGUOUS which allows replication of a datatype into contiguous locations.

MPI\_TYPE\_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (non-negative integer)	34
			35
IN	oldtype	old datatype (handle)	36
OUT	newtype	new datatype (handle)	37
			38
int MPI_T	ype_contiguous(int count,	, MPI_Datatype oldtype,	39
	MPI_Datatype *newtyp	e)	40
NDT T			41
• -	0	e, newtype, ierror) BIND(C)	42
	ER, INTENT(IN) :: count		43
	<pre>MPI_Datatype), INTENT(IN)</pre>		44
	MPI_Datatype), INTENT(OUT		45
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	46
MPI_TYPE_	CONTIGUOUS(COUNT, OLDTYPE	E, NEWTYPE, IERROR)	47
INTEG	ER COUNT, OLDTYPE, NEWTYP	PE, IERROR	48

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

1 2 3	5.		by concatenating <b>count</b> copies of <i>extent</i> as the size of the concatenated copies.
4 5	-	<b>4.2</b> Let oldtype have type ma The type map of the datatype	up $\{(\texttt{double}, 0), (\texttt{char}, 8)\}$ , with extent 16, and let e returned by <b>newtype</b> is
6 7	{(dou	(ble, 0), (char, 8), (double, 16)	$, (\texttt{char}, 24), (\texttt{double}, 32), (\texttt{char}, 40) \};$
8 9 10		ting double and char elemen eral, assume that the type ma	ts, with displacements $0, 8, 16, 24, 32, 40$ . p of oldtype is
11	$\{(typ$	$e_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$	$_{-1})\},$
12 13	with extent	x ex. Then newtype has a type	e map with $count \cdot n$ entries defined by:
14	$\{(typ$	$e_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$	$(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex),$
15 16 17	$\ldots, (a)$	$type_0, disp_0 + ex \cdot (count - 1))$	$,\ldots,(type_{n-1},disp_{n-1}+ex\cdot(count-1))\}.$
18 19 20 21 22	cation of a obtained by	datatype into locations that	OR is a more general constructor that allows repli- consist of equally spaced blocks. Each block is mber of copies of the old datatype. The spacing t of the old datatype.
23 24	MPI_TYPE	_VECTOR(count, blocklength,	stride, oldtype, newtype)
25	IN	count	number of blocks (non-negative integer)
26 27 28	IN	blocklength	number of elements in each block (non-negative integer)
29 30	IN	stride	number of elements between start of each block (integer)
31 32	IN	oldtype	old datatype (handle)
33	OUT	newtype	new datatype (handle)
34 35 36	int MPI_T		blocklength, int stride, MPI_Datatype *newtype)
37 38 39 40 41 42	INTEG TYPE (1 TYPE (1	vector(count, blocklength BIND(C) ER, INTENT(IN) :: count, MPI_Datatype), INTENT(IN) MPI_Datatype), INTENT(OUT ER, OPTIONAL, INTENT(OUT)	:: oldtype ) :: newtype
43 44 45 46 47 48	MPI_TYPE_	VECTOR(COUNT, BLOCKLENGTH	, STRIDE, OLDTYPE, NEWTYPE, IERROR) RIDE, OLDTYPE, NEWTYPE, IERROR

**Example 4.3** Assume, again, that oldtype has type map {(double, 0), (char, 8)}, with extent 16. A call to MPI\_TYPE\_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,

 $\{(\texttt{double},0),(\texttt{char},8),(\texttt{double},16),(\texttt{char},24),(\texttt{double},32),(\texttt{char},40),$ 

 $(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}.$ 

That is, two blocks with three copies each of the old type, with a stride of 4 elements  $(4 \cdot 16)$  bytes) between the the start of each block.

**Example 4.4** A call to MPI\_TYPE\_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

 $\{(double, 0), (char, 8), (double, -32), (char, -24), (double, -64), (char, -56)\}.$ 

In general, assume that oldtype has type map,

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

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with count  $\cdot$  bl  $\cdot$  n entries:

$$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \\ (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, \\ (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex), \\ (type_0, disp_0 + stride \cdot ex), \dots, (type_{n-1}, disp_{n-1} + stride \cdot ex), \dots, \\ (type_0, disp_0 + (stride + bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \dots, \\ (type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \dots, \\ (type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} +$$

A call to MPI\_TYPE\_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI\_TYPE\_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI\_TYPE\_VECTOR(1, count, n, oldtype, newtype), n arbitrary.

```
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
                   stride
        IN
                                                   number of bytes between start of each block (integer)
12
        IN
                   oldtype
                                                   old datatype (handle)
13
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
      MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
19
                       ierror) BIND(C)
20
           INTEGER, INTENT(IN) :: count, blocklength
21
           INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
22
           TYPE(MPI_Datatype), INTENT(IN) :: oldtype
23
           TYPE(MPI_Datatype), INTENT(OUT) :: newtype
24
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
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}
           Assume that oldtype has type map,
32
            \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\
33
      with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
34
      count \cdot bl \cdot n entries:
35
            \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \}
36
37
            (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,
38
            (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
39
40
            (type_0, disp_0 + \mathsf{stride}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \ldots,
41
            (type_0, disp_0 + stride + (bl - 1) \cdot ex), \ldots,
42
43
            (type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex), \ldots,
44
            (type_0, disp_0 + stride \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)), \ldots,
45
46
            (type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex), \ldots,
47
            (type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.
48
```

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.

## MPI\_TYPE\_INDEXED(count, array\_of\_blocklengths, array\_of\_displacements, oldtype, newtype)

	. ,		
IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)	9 10 11
IN	array_of_blocklengths	number of elements per block (array of non-negative integers)	12 13 14
IN	array_of_displacements	displacement for each block, in multiples of oldtype extent (array of integer)	15 16
IN	oldtype	old datatype (handle)	17
OUT	newtype	new datatype (handle)	18 19
int MDT T	vpe indexed(int count co	onst int array_of_blocklengths[], const	20
IIIC MFI_I		ements[], MPI_Datatype oldtype,	21
	MPI_Datatype *newtype		22
			23
MPI_Type_	<pre>indexed(count, array_of_b</pre>	olocklengths, array_of_displacements,	24
	oldtype, newtype, ie:		25
INTEG	ER, INTENT(IN) :: count,	array_of_blocklengths(count),	26

oldtype

ierror

newtype

Example 4.5

Let oldtype have type map  $\{(double, 0), (char, 8)\}$ , with extent 16. Let B = (3, 1) and let D = (4, 0). A call to MPI\_TYPE\_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

 $\{(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104), \}$ 

MPI\_TYPE\_INDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS, ARRAY\_OF\_DISPLACEMENTS,

INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(\*), ARRAY\_OF\_DISPLACEMENTS(\*),

 $(\texttt{double}, 0), (\texttt{char}, 8)\}.$ 

OLDTYPE, NEWTYPE, IERROR

array\_of\_displacements(count)

TYPE(MPI\_Datatype), INTENT(IN) ::

TYPE(MPI\_Datatype), INTENT(OUT) ::

INTEGER, OPTIONAL, INTENT(OUT) ::

OLDTYPE, NEWTYPE, IERROR)

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

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

1

2

3

4

5 6

7

8

27

28

29

30

31

32

33

34

35 36

37

38

39

40 41

42

43 44

45

46

1			locklengths argument and D be the
2 3	array_of_o	displacements argument. The n	Newly created datatype has $n \cdot \sum_{i=0}^{count-1} B[i]$ entries:
4	$\{(ty$	$pe_0, disp_0 + D[0] \cdot ex), \dots, (typ_0)$	$pe_{n-1}, disp_{n-1} + D[0] \cdot ex), \dots,$
5 6	(typ	$e_0, disp_0 + (D[0] + B[0] - 1) \cdot e_0$	$ex), \ldots, (type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), \ldots,$
7 8	(typ	$e_0, disp_0 + D[count-1] \cdot ex), \dots$	$, (type_{n-1}, disp_{n-1} + D[count-1] \cdot ex), \dots,$
9 10	(typ	$e_0, disp_0 + (D[count-1] + B[count-1]]$	$[unt-1]-1)\cdot ex),\ldots,$
11 12	(typ	$e_{n-1}, disp_{n-1} + (D[count-1] +$	$B[count-1] - 1) \cdot ex) \}.$
13 14			nt, blocklength, stride, oldtype, newtype) is equivalent t, B, D, oldtype, newtype) where
15 16	D[j]	$= j \cdot stride, \ j = 0, \dots, count - \mathbf{v}$	- 1,
17	and		
18	D[:]	hla aldan ath 🔅 0 an un	+ 1
19 20	Б[]]	= blocklength, $j = 0, \ldots, $ coun	t = 1.
20	Hindexed	The function MPI TYPE C	REATE_HINDEXED is identical to
22			k displacements in array_of_displacements are spec-
23		tes, rather than in multiples of	
24			
25		E CREATE HINDEYED	, array_of_blocklengths, array_of_displacements,
26		oldtype, newtype)	., array_or_blocklengths, array_or_displacements,
27 28	INI		
29 30	IN	count	<pre>number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non- negative integer)</pre>
31 32	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
33 34	IN	array_of_displacements	byte displacement of each block (array of integer)
34 35	IN	oldtype	old datatype (handle)
36			·- · · ·
37	OUT	newtype	new datatype (handle)
38 39	int MPI_	<i>v</i> 1	<pre>count, const int array_of_blocklengths[], v_of_displacements[], MPI_Datatype oldtype,</pre>
40 41		MPI_Datatype *newtyp	
42	MPT Tune	_create_hindexed(count, a	rray of blocklengths
43	III I_IYPC		nts, oldtype, newtype, ierror) BIND(C)
44	INTE	v 1	, array_of_blocklengths(count)
45		GER(KIND=MPI_ADDRESS_KIND	•
46	arra	y_of_displacements(count)	
47		(MPI_Datatype), INTENT(IN	V -
48	TYPE	(MPI_Datatype), INTENT(OU	T) :: newtype

INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	1
MPI_TYPE_CREATE_HINDEXED(COUNT, AR	RAY_OF_BLOCKLENGTHS,	2
-	'S, OLDTYPE, NEWTYPE, IERROR)	3 4
	ENGTHS(*), OLDTYPE, NEWTYPE, IERROR	5
INTEGER(KIND=MPI_ADDRESS_KIND)	ARRAY_OF_DISPLACEMENTS(*)	6
Assume that oldtype has type map,		7
$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$	.1)},	8 9
with writer to a Let D he the energy of		10
with extent <i>ex</i> . Let B be the array_of_ array of displacements argument. The r	here has a type map with $n \cdot n$	11
$\sum_{i=0}^{\text{count}-1} B[i]$ entries:	lowly created datatype has a type hap with h	12 13
$\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, \dots, type_{n-1}, \dots, t$	dicm = (D[0])	13
$\{(igpe_0, aisp_0 + D[0]), \ldots, (igpe_{n-1}, d)\}$	$(uisp_{n-1} + D[0]), \ldots,$	15
$(type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex$	$(x),\ldots,$	16
$(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1)$	(), ar)	17 18
$(lgpe_{n-1}, alsp_{n-1} + D[0] + (D[0] - 1))$	$() \cdot e_{x}), \ldots,$	18
$(type_0, disp_0 + D[count-1]), \dots, (typ)$	$pe_{n-1}, disp_{n-1} + D[count-1]), \dots,$	20
$(type_0, disp_0 + D[count-1] + (B[cour])$	(1, 1)	21
$(igpe_0, aisp_0 + D[count-1] + (D[count-1]))$	$[\mathbf{L}^{-1}] = 1 / (\mathcal{E}_{\mathcal{L}}), \dots,$	22
$(type_{n-1}, disp_{n-1} + D[count-1] + (B$	$[count-1] - 1) \cdot ex) \}.$	23 24
		25
	e as MPI_TYPE_INDEXED except that the block-	26
0	are many codes using indirect addressing arising size is always 1 (gather/scatter). The following	27
convenience function allows for constant		28
		29 30
MDI TYDE CREATE INDEVED DI OCIA	(acust blacklangth away of displacements oldtyre	31
newtype)	(count, blocklength, array_of_displacements, oldtype,	32
<u> </u>	length of array of displacements (non-negative integer)	33
		34
IN blocklength	size of block (non-negative integer)	35 36
IN array_of_displacements	array of displacements (array of integer)	37
IN oldtype	old datatype (handle)	38
OUT newtype	new datatype (handle)	39
		$40 \\ 41$
<pre>int MPI_Type_create_indexed_block()</pre>	-	41
int array_of_displace MPI_Datatype *newtype	<pre>ments[], MPI_Datatype oldtype, )</pre>	43
		44

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
\mathbf{2}
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
5
                    OLDTYPE, NEWTYPE, IERROR)
6
          INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
7
         NEWTYPE, IERROR
8
9
10
     Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to
11
     MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in
12
     array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.
13
14
     MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements,
15
16
                    oldtype, newtype)
17
       IN
                count
                                            length of array of displacements (non-negative integer)
18
       IN
                blocklength
                                            size of block (non-negative integer)
19
                array_of_displacements
       IN
                                           byte displacement of each block (array of integer)
20
21
       IN
                oldtype
                                            old datatype (handle)
22
       OUT
                newtype
                                           new datatype (handle)
23
^{24}
     int MPI_Type_create_hindexed_block(int count, int blocklength, const
25
                    MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
26
                    MPI_Datatype *newtype)
27
28
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
29
                    oldtype, newtype, ierror) BIND(C)
30
          INTEGER, INTENT(IN) :: count, blocklength
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
32
         array_of_displacements(count)
33
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
34
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
37
                    OLDTYPE, NEWTYPE, IERROR)
38
         INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
39
          INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
40
41
42
     Struct MPI_TYPE_CREATE_STRUCT is the most general type constructor. It further
43
     generalizes MPI_TYPE_CREATE_HINDEXED in that it allows each block to consist of repli-
44
     cations of different datatypes.
45
46
47
48
```

MPI_	TYPE_CREATE_STRUCT(count, array_of_types, newtype	array_of_blocklengths, array_of_displacements,	$\frac{1}{2}$
IN	count	number of blocks (non-negative integer) — also num- ber of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths	3 4 5 6
IN	array_of_blocklength	number of elements in each block (array of non-negative integer)	7 8
IN	array_of_displacements	byte displacement of each block (array of integer)	9
IN	array_of_types	type of elements in each block (array of handles to datatype objects)	10 11 12
OU	T newtype	new datatype (handle)	12 13 14
int 1	const MPI_Aint arra	<pre>punt, const int array_of_blocklengths[], y_of_displacements[], const of_types[], MPI_Datatype *newtype)</pre>	15 16 17
MPI_	Type_create_struct(count, ar: array_of_displaceme BIND(C)	ray_of_blocklengths, nts, array_of_types, newtype, ierror)	18 19 20
		t, array_of_blocklengths(count) D), INTENT(IN) ::	21 22
	array_of_displacements(count		23 24
	<pre>TYPE(MPI_Datatype), INTENT(I)</pre>		24 25
	TYPE(MPI_Datatype), INTENT(O		26
	INTEGER, OPTIONAL, INTENT(OU	I) :: lerror	27
MPI_	TYPE_CREATE_STRUCT(COUNT, AR		28
	INTEGER COUNT, ARRAY_OF_BLOC	NTS, ARRAY_OF_TYPES, NEWTYPE, IERROR) KLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,	29 30
	IERROR INTEGER(KIND=MPI_ADDRESS_KIN		31 32
	INTEGER (KIND-MPI_ADDRESS_KIN	D) ARRAI_OF_DISPLACEMENTS(*)	33
E .			34
Exai	<b>nple 4.6</b> Let type1 have type ma	up,	35
	$\{(\texttt{double}, 0), (\texttt{char}, 8)\},$		36
ith	$rate 16$ Let $P_{1}(2, 1, 2) D_{2}(6)$	16 26) and T (MDI FLOAT time1 MDI CHAD)	37
		), 16, 26), and $T = (MPI_FLOAT, type1, MPI_CHAR)$ . FRUCT(3, B, D, T, newtype) returns a datatype with	38
type		(i) D, D, T, newtype) returns a datatype with	39 40
J 1			40
	$\{(\texttt{float}, 0), (\texttt{float}, 4), (\texttt{double}, 1)\}$	$6), (\texttt{char}, 24), (\texttt{char}, 26), (\texttt{char}, 27), (\texttt{char}, 28) \}.$	42
That	is, two copies of MPI_FLOAT sta	rting at 0, followed by one copy of type1 starting at	43
		AR, starting at 26. (We assume that a float occupies	44
	bytes.)		45
]	n general, let T be the array_of_t	ypes argument, where T[i] is a handle to,	46
	$typemap_i = \{(type_0^i, disp_0^i), \dots, (a_{i}^{i})\}$	$type_{n_i-1}^i, disp_{n_i-1}^i)\},$	47 48

1	with exten	t $ex$ : Let B be the array of	-blocklength argument and D be the
2	array_of_d	splacements argument. Let c	be the count argument. Then the newly created
3	datatype h	as a type map with $\sum_{i=0}^{C-1} B[i]$	$\cdot n_i$ entries:
4 5	$\{(typ$	$pe_0^0, disp_0^0 + D[0]), \dots, (type_{n_0}^0, type_{n_0}^0)$	$disp_{n_0}^0 + D[0]), \dots,$
6 7	(type	${}^{0}_{0}, disp^{0}_{0} + D[0] + (B[0] - 1) \cdot e$	$(x_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0]-1) \cdot ex_0), \dots,$
8 9	(type	$\mathbf{C}_{0}^{c-1}, disp_{0}^{c-1} + D[c-1]), \dots, (ty)$	$pe_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1]), \dots,$
10 11	(type	$B_0^{C-1}, disp_0^{C-1} + D[c-1] + (B[c-1])$	$]-1) \cdot ex_{C-1}), \ldots,$
12 13	(type	$L_{n_{C-1}-1}^{c-1}$ , $disp_{n_{C-1}-1}^{c-1}$ + D[c-1] +	$(B[c-1]-1) \cdot ex_{C-1})\}.$
14 15			DEXED(count, B, D, oldtype, newtype) is equivalent CT(count, B, D, T, newtype), where each entry of T
16 17	is equal to		
18		5	
19	4.1.3 Sul	parray Datatype Constructor	
20 21			
21 22 23	MPI_TYPE	E_CREATE_SUBARRAY(ndims order, oldtype, newtype)	, array_of_sizes, array_of_subsizes, array_of_starts,
24	IN	ndims	number of array dimensions (positive integer)
25 26 27	IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)
28 29	IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)
30 31	IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)
32	IN	order	array storage order flag (state)
33 34	IN	oldtype	array element datatype (handle)
35	OUT	newtype	new datatype (handle)
36			
37 38	int MPI_T	· · · · ·	ndims, const int array_of_sizes[], const
39		Ũ	s[], const int array_of_starts[], int oldtype, MPI_Datatype *newtype)
40			
41 42	MPI_Type_	•	rray_of_sizes, array_of_subsizes, er, oldtype, newtype, ierror) BIND(C)
42	INTEG	ER, INTENT(IN) :: ndims,	
44			ay_of_starts(ndims), order
45		MPI_Datatype), INTENT(IN)	
46		MPI_Datatype), INTENT(OUT	V1
47	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror
48			

The subarray type constructor creates an MPI datatype describing an *n*-dimensional subarray of an *n*-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1 on page 489.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array\_of\_sizes, array\_of\_subsizes, and array\_of\_starts.

The number of elements of type oldtype in each dimension of the *n*-dimensional array and the requested subarray are specified by array\_of\_sizes and array\_of\_subsizes, respectively. For any dimension i, it is erroneous to specify array\_of\_subsizes[i] < 1 or array\_of\_subsizes[i] > array\_of\_sizes[i].

The array\_of\_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to specify array\_of\_starts[i] < 0 or array\_of\_starts[i] > (array\_of\_sizes[i] - array\_of\_subsizes[i]).

Advice to users. In a Fortran program with arrays indexed starting from 1, if the 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.*)

The order argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

**MPI\_ORDER\_C** The ordering used by C arrays, (i.e., row-major order)

**MPI\_ORDER\_FORTRAN** The ordering used by Fortran arrays, (i.e., column-major order)

A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:

newtype	=	Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },	37
		$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$	38
			39
		$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	40

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. These equations

 $\overline{7}$ 

1 2 3	use the conceptual data types $lb\_marker$ and $ub\_marker$ , see Section 4.1.6 on page 103 for details.
4	
5	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, $ $(4.2)$
6	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$
7 8	$= \{(lb\_marker, 0),$
9	$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$
10	$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$
11	$disp_{n-1} + (start_0 + 1) \times ex), \dots$
12	$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,$
13	$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$
14 15	$(ub\_marker, size_0 \times ex)$ }
16	
17	$Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\},$ (4.3)
18	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \{subsize_0, subsize_1, \dots, subsize_{ndims-1}\}, \{subsize_1, \dots, subsize_{ndims-1}\}, \{subsize_{ndims-1}\}, \{subsize_{ndi$
19	$\{start_0, start_1, \dots, start_{ndims-1}\}, $ oldtype)
20	$= \text{Subarray}(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\}, $
21 22	
23	$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},$
24	$\{start_1, start_2, \dots, start_{ndims-1}\},\$
25	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$
26	
27	Subarray( $ndims$ , { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.4)
28	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$
29 30	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$
31	= Subarray( $ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$
32	$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},\$
33	$\{start_0, start_1, \ldots, start_{ndims-2}\},\$
34	Subarray $(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$
35	
36	For an example use of MPI TYPE CREATE SUBARRAY in the context of I/O see Sec-

<sup>30</sup> For an example use of MPI\_TYPE\_CREATE\_SUBARRAY in the context of I/O see Sec-<sup>37</sup> tion 13.11.2.

### 4.1.4 Distributed Array Datatype Constructor

39

40

43

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.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI\_FILE\_SET\_VIEW), see MPI I/O, especially Section 13.1.1

#### 4.1. DERIVED DATATYPES

on page 489 and Section 13.3 on page 501. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

# MPI\_TYPE\_CREATE\_DARRAY(size, rank, ndims, array\_of\_gsizes, array\_of\_distribs, array\_of\_dargs, array\_of\_psizes, order, oldtype, newtype)

IN	size	size of process group (positive integer)	9
IN	rank	rank in process group (non-negative integer)	10
IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)	11 12 13
IN	array_of_gsizes	number of elements of type <b>oldtype</b> in each dimension of global array (array of positive integers)	14 15
IN	array_of_distribs	distribution of array in each dimension (array of state)	16
IN	array_of_dargs	distribution argument in each dimension (array of pos- itive integers)	17 18 19
IN	array_of_psizes	size of process grid in each dimension (array of positive integers)	20 21
IN	order	array storage order flag (state)	22
IN	oldtype	old datatype (handle)	23 24
OUT	newtype	new datatype (handle)	24
	51		26
	int array_of_gsizes[ int array_of_dargs[] MPI_Datatype oldtype	<pre>ze, int rank, int ndims, const ], const int array_of_distribs[], const , const int array_of_psizes[], int order, e, MPI_Datatype *newtype)</pre>	28 29 30 31
INTE( array array TYPE TYPE	array_of_distribs, a oldtype, newtype, ie	<pre>rank, ndims, array_of_gsizes(ndims), ay_of_dargs(ndims), ) :: oldtype T) :: newtype</pre>	32 33 34 35 36 37 38 39 40
INTEG	ARRAY_OF_DISTRIBS, A OLDTYPE, NEWTYPE, IE GER SIZE, RANK, NDIMS, AR	, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, ERROR) RAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*), SIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR	41 42 43 44 45
		be used to generate the datatypes corresponding to array of <b>oldtype</b> elements onto an <b>ndims</b> -dimensional	46 47 48

97

1

 $\mathbf{2}$ 

 $\overline{7}$ 

1	grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See
2	Example 4.7 on page 101.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the
3	equation $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$ must be satisfied. The ordering of processes
4	in the process grid is assumed to be row-major, as in the case of virtual Cartesian process
5	topologies.
6	Advice to users. For both Fortran and C arrays, the ordering of processes in the
7	
8	process grid is assumed to be row-major. This is consistent with the ordering used in
9	virtual Cartesian process topologies in MPI. To create such virtual process topologies,
10	or to find the coordinates of a process in the process grid, etc., users may use the
11	corresponding process topology functions, see Chapter 7 on page 289. (End of advice
12	to users.)
13 14	Each dimension of the array can be distributed in one of three ways:
15	• MPI_DISTRIBUTE_BLOCK - Block distribution
16 17	• MPI_DISTRIBUTE_CYCLIC - Cyclic distribution
18 19	• MPI_DISTRIBUTE_NONE - Dimension not distributed.
20	The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument.
20	The distribution argument for a dimension that is not distributed is ignored. For any
22	dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify
22	array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].
23	For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to
24 25	MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-
26	RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of
20	MPI_DISTRIBUTE_DFLT_DARG.
28	The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the stor-
29	age order. Therefore, arrays described by this type constructor may be stored in Fortran
30	(column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN
31	and MPI_ORDER_C.
32	This routine creates a new MPI datatype with a typemap defined in terms of a function
33	called "cyclic()" (see below).
	Without loss of generality, it suffices to define the typemap for the
34	MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.
35	MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the
36	MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.
37	MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG
38	is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to
39	is equivalent to MFI_DISTRIBUTE_CTCLIC with anay_of_dargs[i] set to
40 41	$(array_of_gsizes[i] + array_of_psizes[i] - 1)/array_of_psizes[i].$
42	If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and
43	MPI_DISTRIBUTE_CYCLIC are equivalent.
44	MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with
45	array_of_dargs[i] set to array_of_gsizes[i].
46	Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to
47	MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with
48	array_of_dargs[i] set to 1.

For MPI\_ORDER\_FORTRAN, an ndims-dimensional distributed array (newtype) is defined by the following code fragment:

```
oldtypes[0] = oldtype;
                                                                                               4
    for (i = 0; i < ndims; i++) {</pre>
                                                                                               5
         oldtypes[i+1] = cyclic(array_of_dargs[i],
                                                                                               6
                                     array_of_gsizes[i],
                                                                                               7
                                     r[i],
                                                                                               8
                                     array_of_psizes[i],
                                                                                               9
                                     oldtypes[i]);
                                                                                               10
    }
                                                                                               11
    newtype = oldtypes[ndims];
                                                                                               12
                                                                                               13
    For MPI_ORDER_C, the code is:
                                                                                               14
                                                                                               15
    oldtypes[0] = oldtype;
                                                                                               16
    for (i = 0; i < ndims; i++) {</pre>
                                                                                               17
         oldtypes[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
                                                                                               18
                                       array_of_gsizes[ndims - i - 1],
                                                                                               19
                                       r[ndims - i - 1],
                                                                                               20
                                       array_of_psizes[ndims - i - 1],
                                                                                               21
                                       oldtypes[i]);
                                                                                               22
    }
                                                                                               23
    newtype = oldtypes[ndims];
                                                                                               ^{24}
                                                                                               25
                                                                                               26
where r[i] is the position of the process (with rank rank) in the process grid at dimension i.
                                                                                               27
The values of r[i] are given by the following code fragment:
                                                                                               28
                                                                                               29
    t_rank = rank;
                                                                                               30
    t_size = 1;
                                                                                               31
    for (i = 0; i < ndims; i++)</pre>
                                                                                               32
         t_size *= array_of_psizes[i];
                                                                                               33
    for (i = 0; i < ndims; i++) {</pre>
                                                                                               34
         t_size = t_size / array_of_psizes[i];
                                                                                               35
         r[i] = t_rank / t_size;
                                                                                               36
         t_rank = t_rank % t_size;
                                                                                               37
    }
                                                                                               38
Let the typemap of oldtype have the form:
                                                                                               39
                                                                                               40
     \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
                                                                                               41
                                                                                               42
where type_i is a predefined MPI datatype, and let ex be the extent of oldtype. The following
                                                                                               43
function uses the conceptual datatypes lb_marker and ub_marker, see Section 4.1.6 on
                                                                                               44
page 103 for details.
                                                                                               45
    Given the above, the function cyclic() is defined as follows:
                                                                                               46
     cyclic(darg, gsize, r, psize, oldtype)
                                                                                               47
                                                                                               48
       = \{(lb_marker, 0),
```

1

2

1	$(type_0, disp_0 + r \times darg \times ex), \ldots,$
2	$(type_{n-1}, disp_{n-1} + r \times darg \times ex),$
3	$(type_0, disp_0 + (r \times darg + 1) \times ex), \dots,$
4	$(type_{n-1}, disp_{n-1} + (r \times darq + 1) \times ex), \dots, (type_{n-1}, disp_{n-1} + (r \times darq + 1) \times ex),$
5	$(igpe_{n-1}, uisp_{n-1} + (i \land uuig + 1) \land ex),$
6 7	(1, 1) + ((1, 1) + 1 + 1)
8	$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \dots,$
9	$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),$
10	
11	$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \dots,$
12	$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$
13	$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,$
14 15	$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$
16	
17	$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$
18	$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),$
19	$(sg_{F}s_{n-1}, ssg_{F}n_{-1} + ((r+1) + ssg_{F}g_{F}) + ssg_{F}g_{F}s_{n-1})$
20	:
21	$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$
22	$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$
23 24	$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,$
25	$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$
26	$+psize \times darg \times ex \times (count - 1)),$
27	•••
28	$(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$
29	$+psize  imes darg  imes ex  imes (count - 1)), \dots,$
30	$(type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex$
31 32	$+psize \times darg \times ex \times (count - 1)),$
33	
34	$(ub\_marker,gsize * ex)\}$
35	where $count$ is defined by this code fragment:
36	<pre>nblocks = (gsize + (darg - 1)) / darg;</pre>
37	count = nblocks / psize;
38	<pre>left_over = nblocks - count * psize;</pre>
39 40	if (r < left_over)
41	<pre>count = count + 1;</pre>
42	Here, <i>nblocks</i> is the number of blocks that must be distributed among the processors.
43	Finally, $darg_{last}$ is defined by this code fragment:
44	
45	if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
46	<pre>darg_last = darg; alao {</pre>
47 48	<pre>else {     darg_last = num_in_last_cyclic - darg * r;</pre>
10	aug_idee nam_in_idee_ojeite aug i,

if	(darg_last > darg)	1
	darg_last = darg;	2
if	(darg_last <= 0)	3
	<pre>darg_last = darg;</pre>	4
}		5

**Example 4.7** Consider generating the filetypes corresponding to the HPF distribution:

```
<oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

This can be achieved by the following Fortran code, assuming there will be six processes attached to the run:

```
ndims = 3
array_of_gsizes(1) = 100
array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
\operatorname{array_of_dargs}(1) = 10
array_of_gsizes(2) = 200
array_of_distribs(2) = MPI_DISTRIBUTE_NONE
\operatorname{array_of_dargs}(2) = 0
array_of_gsizes(3) = 300
array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
array_of_psizes(1) = 2
array_of_psizes(2) = 1
array_of_psizes(3) = 3
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
     array_of_distribs, array_of_dargs, array_of_psizes,
                                                                    &
     MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

#### 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.

The address of a location in memory can be found by invoking the function MPI\_GET\_ADDRESS.

 $^{24}$ 

```
1
     MPI_GET_ADDRESS(location, address)
2
       IN
                 location
                                             location in caller memory (choice)
3
       OUT
                 address
                                             address of location (integer)
4
5
6
     int MPI_Get_address(const void *location, MPI_Aint *address)
\overline{7}
     MPI_Get_address(location, address, ierror) BIND(C)
8
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
9
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
13
          <type> LOCATION(*)
14
          INTEGER IERROR
15
          INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
16
          Returns the (byte) address of location.
17
18
                              Current Fortran MPI codes will run unmodified, and will port
           Advice to users.
19
           to any system. However, they may fail if addresses larger than 2^{32} - 1 are used
20
           in the program. New codes should be written so that they use the new functions.
21
           This provides compatibility with C/C++ and avoids errors on 64 bit architectures.
22
           However, such newly written codes may need to be (slightly) rewritten to port to old
23
           Fortran 77 environments that do not support KIND declarations. (End of advice to
24
           users.)
25
26
                         In the mpi_f08 module, the location argument is not defined with
           Rationale.
27
           INTENT(IN) because existing applications may use MPI_GET_ADDRESS as a substi-
28
           tute for MPI_F_SYNC_REG that was not defined before MPI-3.0. (End of rationale.)
29
30
     Example 4.8 Using MPI_GET_ADDRESS for an array.
31
32
        REAL A(100,100)
33
34
         INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
        CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
35
        CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
36
        DIFF = I2 - I1
37
     ! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
38
     ! implementation dependent.
39
40
                              C users may be tempted to avoid the usage of
           Advice to users.
41
           MPI_GET_ADDRESS and rely on the availability of the address operator &. Note,
42
           however, that & cast-expression is a pointer, not an address. ISO C does not require
43
           that the value of a pointer (or the pointer cast to int) be the absolute address of the
44
           object pointed at — although this is commonly the case. Furthermore, referencing
45
           may not have a unique definition on machines with a segmented address space. The
46
           use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such
47
           machines as well. (End of advice to users.)
```

#### 4.1. DERIVED DATATYPES

1

 $\mathbf{2}$ 

3

4

56 7

8

9

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10-17.1.20. (End of advice to users.) The following auxiliary functions provide useful information on derived datatypes. MPI\_TYPE\_SIZE(datatype, size) IN datatype datatype (handle) 10 OUT datatype size (integer) size 11 12int MPI\_Type\_size(MPI\_Datatype datatype, int \*size) 13 14MPI\_Type\_size(datatype, size, ierror) BIND(C) 15TYPE(MPI\_Datatype), INTENT(IN) :: datatype 16INTEGER, INTENT(OUT) :: size 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI\_TYPE\_SIZE(DATATYPE, SIZE, IERROR) 19 INTEGER DATATYPE, SIZE, IERROR 202122 MPI\_TYPE\_SIZE\_X(datatype, size) 2324IN datatype datatype (handle) 25OUT size datatype size (integer) 2627int MPI\_Type\_size\_x(MPI\_Datatype datatype, MPI\_Count \*size) 28 29 MPI\_Type\_size\_x(datatype, size, ierror) BIND(C) 30 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 31 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: size 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 MPI\_TYPE\_SIZE\_X(DATATYPE, SIZE, IERROR) 34 INTEGER DATATYPE, IERROR 35 INTEGER(KIND = MPI\_COUNT\_KIND) SIZE 36 37

MPI\_TYPE\_SIZE and MPI\_TYPE\_SIZE\_X set the value of size to the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity. For both functions, if the 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.

#### Lower-Bound and Upper-Bound Markers 4.1.6

It is often convenient to define explicitly the lower bound and upper bound of a type map, 46and override the definition given on page 104. This allows one to define a datatype that has 47"holes" at its beginning or its end, or a datatype with entries that extend above the upper 48

43 4445

38

39

40

41

. .

<sup>1</sup> bound or below the lower bound. Examples of such usage are provided in Section 4.1.14.
 <sup>2</sup> Also, the user may want to overide the alignment rules that are used to compute upper
 <sup>3</sup> bounds and extents. E.g., a C compiler may allow the user to overide default alignment
 <sup>4</sup> rules for some of the structures within a program. The user has to specify explicitly the
 <sup>5</sup> bounds of the datatypes that match these structures.

<sup>6</sup> To achieve this, we add two additional conceptual datatypes, *lb\_marker* and *ub\_marker*, <sup>7</sup> that represent the lower bound and upper bound of a datatype. These conceptual datatypes <sup>8</sup> occupy no space (*extent*(*lb\_marker*) = *extent*(*ub\_marker*) = 0). They do not affect the <sup>9</sup> size or count of a datatype, and do not affect the content of a message created with this <sup>10</sup> datatype. However, they do affect the definition of the extent of a datatype and, therefore, <sup>11</sup> affect the outcome of a replication of this datatype by a datatype constructor.

12

22 23

25

26

27

29

30 31 32

33 34

42

43

**Example 4.9** A call to MPI\_TYPE\_CREATE\_RESIZED(MPI\_INT, -3, 9, type1) creates a 13 new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer 14at displacement 0. This is the datatype defined by the typemap  $\{(lb_marker, -3), (int, 0), \}$ 15(ub\_marker, 6)}. If this type is replicated twice by a call to MPI\_TYPE\_CONTIGUOUS(2, 16type1, type2) then the newly created type can be described by the typemap  $\{(lb_marker,$ 17-3), (int, 0), (int, 9), (ub\_marker, 15)}. (An entry of type ub\_marker can be deleted if there 18 is another entry of type *ub\_marker* with a higher displacement; an entry of type *lb\_marker* 19can be deleted if there is another entry of type *lb\_marker* with a lower displacement.) 20In general, if 21

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

 $_{24}$  then the *lower bound* of *Typemap* is defined to be

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

 $^{28}$  Similarly, the *upper bound* of *Typemap* is defined to be

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

Then

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

<sup>35</sup> If  $type_i$  requires alignment to a byte address that is a multiple of  $k_i$ , then  $\epsilon$  is the least <sup>36</sup> non-negative increment needed to round extent(Typemap) to the next multiple of max<sub>i</sub>  $k_i$ . <sup>37</sup> In Fortran, it is implementation dependent whether the MPI implementation computes <sup>39</sup> the alignments  $k_i$  according to the alignments used by the compiler in common blocks, <sup>40</sup> SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE <sup>41</sup> nor BIND(C).

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

<sup>44</sup> Rationale. Before Fortran 2003, MPI\_TYPE\_CREATE\_STRUCT could be applied to <sup>45</sup> Fortran common blocks and SEQUENCE derived types. With Fortran 2003, this list <sup>46</sup> was extended by BIND(C) derived types and MPI implementors have implemented the <sup>47</sup> alignments  $k_i$  differently, i.e., some based on the alignments used in SEQUENCE derived <sup>48</sup> 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.)

Advice to users. Structures combining different basic datatypes should be defined so that there will be no gaps based on alignment rules. If such a datatype is used to create an array of structures, users should also avoid an alignment-gap at the end of the structure. In MPI communication, the content of such gaps would not be communicated into the receiver's buffer. For example, such an alignment-gap may occur between an odd number of floats or REALs before a double or DOUBLE 11 PRECISION data. Such gaps may be added explicitly to both the structure and the MPI 12derived datatype handle because the communication of a contiguous derived datatype 13 may be significantly faster than the communication of one that is non-contiguous because of such alignment-gaps.

Example: Instead of

```
TYPE, BIND(C) :: my_data
  REAL, DIMENSION(3) :: x
  ! there may be a gap of the size of one REAL
  ! if the alignment of a DOUBLE PRECISION is
  ! two times the size of a REAL
  DOUBLE PRECISION :: p
END TYPE
```

one should define

```
TYPE, BIND(C) :: my_data
 REAL, DIMENSION(3) :: x
 REAL :: gap1
 DOUBLE PRECISION :: p
END TYPE
```

and also include gap1 in the matching MPI derived datatype. It is required that all processes in a communication add the same gaps, i.e., defined with the same basic datatype. Both the original and the modified structures are portable, but may have different performance implications for the communication and memory accesses during computation on systems with different alignment values.

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a  $max_ik_i$  alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 17.1.15 on page 633. (End of advice to users.)

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

14

1516

17 18

19

20

21

22

23

 $^{24}$ 

2526

2728

29

30

31

32

33 34 35

36

37

38

39

40

41

42

43

44

```
CHAPTER 4. DATATYPES
1
     4.1.7
            Extent and Bounds of Datatypes
\mathbf{2}
3
4
     MPI_TYPE_GET_EXTENT(datatype, lb, extent)
5
       IN
                 datatype
                                             datatype to get information on (handle)
6
       OUT
                 lb
7
                                             lower bound of datatype (integer)
8
       OUT
                                             extent of datatype (integer)
                 extent
9
10
     int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
11
                    MPI_Aint *extent)
12
     MPI_Type_get_extent(datatype, lb, extent, ierror) BIND(C)
13
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
15
16
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
18
          INTEGER DATATYPE, IERROR
19
          INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
20
21
22
     MPI_TYPE_GET_EXTENT_X(datatype, lb, extent)
23
24
       IN
                 datatype
                                             datatype to get information on (handle)
25
       OUT
                 lb
                                             lower bound of datatype (integer)
26
       OUT
                 extent
                                             extent of datatype (integer)
27
28
     int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *1b,
29
                    MPI_Count *extent)
30
^{31}
     MPI_Type_get_extent_x(datatype, lb, extent, ierror) BIND(C)
32
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
          INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
34
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
35
36
     MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
37
          INTEGER DATATYPE, IERROR
          INTEGER(KIND = MPI_COUNT_KIND) LB, EXTENT
38
39
         Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on
40
     page 103).
41
         For both functions, if either OUT parameter cannot express the value to be returned
42
     (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.
43
          MPI allows one to change the extent of a datatype, using lower bound and upper bound
44
     markers. This provides control over the stride of successive datatypes that are replicated
45
     by datatype constructors, or are replicated by the count argument in a send or receive call.
46
47
```

106

MPI_TYPE	E_CREATE_RESIZED(oldtype,	lb, extent, newtype)	1
IN	oldtype	input datatype (handle)	2
IN	lb	new lower bound of datatype (integer)	3 4
IN	extent	new extent of datatype (integer)	4 5
			6
OUT	newtype	output datatype (handle)	7
int MDT T	where the second (MDT D	totune aldterne MDT Aint lb MDT Aint	8
int MPI_I	extent, MPI_Datatype	atatype oldtype, MPI_Aint lb, MPI_Aint	9
	extent, mi_Datatype	*newcype)	10
• 1	• 1	lb, extent, newtype, ierror) BIND(C)	11 12
		), INTENT(IN) :: lb, extent	12
	MPI_Datatype), INTENT(IN MPI_Datatype), INTENT(OU		14
	ER, OPTIONAL, INTENT(OUT)		15
			16
		LB, EXTENT, NEWTYPE, IERROR)	17
	ER OLDTYPE, NEWTYPE, IER ER(KIND=MPI_ADDRESS_KIND		18
INIEG	ER(KIND-HFI_ADDRESS_KIND)	/ LD, EATENT	19

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb + extent. Any previous *lb* and *ub* markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

#### True Extent of Datatypes 4.1.8

Suppose we implement gather (see also Section 5.5 on page 149) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent, for example by using MPI\_TYPE\_CREATE\_RESIZED. The functions MPI\_TYPE\_GET\_TRUE\_EXTENT and MPI\_TYPE\_GET\_TRUE\_EXTENT\_X are provided which return the true extent of the datatype.

MDI TYDE CET TRUE EXTENT(datatura two lh two estant)			
MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)			39
IN	datatype	datatype to get information on (handle)	40
OUT	true_lb	true lower bound of datatype (integer)	41
OUT	true extent	true size of datatype (integer)	42
001	tide_extent	true size of datatype (integer)	43
<pre>int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,</pre>			
MPI_Aint *true_extent)			
MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) BIND(C) 4			

 $^{24}$ 

 $^{31}$ 

1 TYPE(MPI\_Datatype), INTENT(IN) :: datatype  $\mathbf{2}$ INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: true\_lb, true\_extent 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI\_TYPE\_GET\_TRUE\_EXTENT(DATATYPE, TRUE\_LB, TRUE\_EXTENT, IERROR) 5INTEGER DATATYPE, IERROR 6 INTEGER(KIND = MPI\_ADDRESS\_KIND) TRUE\_LB, TRUE\_EXTENT 7 8 9 MPI\_TYPE\_GET\_TRUE\_EXTENT\_X(datatype, true\_lb, true\_extent) 10 11IN datatype datatype to get information on (handle) 12OUT true\_lb true lower bound of datatype (integer) 13 OUT true\_extent true size of datatype (integer) 141516int MPI\_Type\_get\_true\_extent\_x(MPI\_Datatype datatype, MPI\_Count \*true\_lb, 17MPI\_Count \*true\_extent) 18 MPI\_Type\_get\_true\_extent\_x(datatype, true\_lb, true\_extent, ierror) BIND(C) 19TYPE(MPI\_Datatype), INTENT(IN) :: datatype 20INTEGER(KIND = MPI\_COUNT\_KIND), INTENT(OUT) :: true\_lb, true\_extent 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2223MPI\_TYPE\_GET\_TRUE\_EXTENT\_X(DATATYPE, TRUE\_LB, TRUE\_EXTENT, IERROR) 24INTEGER DATATYPE, IERROR 25INTEGER(KIND = MPI\_COUNT\_KIND) TRUE\_LB, TRUE\_EXTENT 26true\_lb returns the offset of the lowest unit of store which is addressed by the datatype, 27i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound mark-28ers. true\_extent returns the true size of the datatype, i.e., the extent of the correspond-29 ing typemap, ignoring explicit lower bound and upper bound markers, and performing no 30 rounding for alignment. If the typemap associated with datatype is  $^{31}$ 32  $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$ 33 34Then 35  $true\_lb(Typemap) = min_i \{ disp_i : type_i \neq lb\_marker, ub\_marker \},$ 36 37  $true\_ub(Typemap) = max_i \{ disp_i + sizeof(type_i) : type_i \neq lb\_marker, ub\_marker \},\$ 38 39 and 40 41  $true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$ 42(Readers should compare this with the definitions in Section 4.1.6 on page 103 and Sec-43 tion 4.1.7 on page 106, which describe the function MPI\_TYPE\_GET\_EXTENT.) 44 The true\_extent is the minimum number of bytes of memory necessary to hold a 45datatype, uncompressed. 46 For both functions, if either OUT parameter cannot express the value to be returned 47(e.g., if the parameter is too small to hold the output value), it is set to MPI\_UNDEFINED. 48

	mmit and Free
A datatyp	e object has to be <i>committed</i> before it can be used in a communication. As
an argume	nt in datatype constructors, uncommitted and also committed datatypes can be
	re is no need to commit basic datatypes. They are "pre-committed."
MPI_TYPI	E_COMMIT(datatype)
INOUT	datatype datatype that is committed (handle)
int MPI_7	Cype_commit(MPI_Datatype *datatype)
MPT Turne	_commit(datatype, ierror) BIND(C)
• -	(MPI_Datatype), INTENT(INOUT) :: datatype
	ER, OPTIONAL, INTENT(OUT) :: ierror
	Let, of Home, Internation, in Forton
	COMMIT(DATATYPE, IERROR)
INTEC	ER DATATYPE, IERROR
The c	ommit operation commits the datatype, that is, the formal description of a com-
	buffer, not the content of that buffer. Thus, after a datatype has been commit-
	be repeatedly reused to communicate the changing content of a buffer or, indeed,
,	t of different buffers, with different starting addresses.
	of anterent surrers, with anterent starting addresses.
Advi	ce to implementors. The system may "compile" at commit time an internal
	esentation for the datatype that facilitates communication, e.g., change from a
-	pacted representation to a flat representation of the datatype, and select the most
-	enient transfer mechanism. (End of advice to implementors.)
MPI_	<b>TYPE_COMMIT</b> will accept a committed datatype; in this case, it is equivalent
to a no-op	
Example	<b>4.10</b> The following code fragment gives examples of using MPI_TYPE_COMMIT.
INTEGER +	zype1, type2
	TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
очгт 1,1L T	! new type object created
CALL MPT	TYPE_COMMIT(type1, ierr)
- III I I I I I I I I I I I I I I I I I	! now type1 can be used for communication
type2 = t	
JP02 (	yper ! type2 can be used for communication
	! (it is a handle to same object as type1)
	TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
CALL MPT	
CALL MPI_	I new uncommitted type object created
	! new uncommitted type object created TYPE COMMIT(type1_ierr)
	TYPE_COMMIT(type1, ierr)
	TYPE_COMMIT(type1, ierr)
	TYPE_COMMIT(type1, ierr)
	TYPE_COMMIT(type1, ierr)

```
1
     MPI_TYPE_FREE(datatype)
\mathbf{2}
       INOUT
                 datatype
                                             datatype that is freed (handle)
3
4
     int MPI_Type_free(MPI_Datatype *datatype)
5
6
     MPI_Type_free(datatype, ierror) BIND(C)
7
          TYPE(MPI_Datatype), INTENT(INOUT) ::
                                                     datatype
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_TYPE_FREE(DATATYPE, IERROR)
10
          INTEGER DATATYPE, IERROR
11
12
          Marks the datatype object associated with datatype for deallocation and sets datatype
13
     to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will
14
     complete normally. Freeing a datatype does not affect any other datatype that was built
15
     from the freed datatype. The system behaves as if input datatype arguments to derived
16
     datatype constructors are passed by value.
17
18
           Advice to implementors. The implementation may keep a reference count of active
19
           communications that use the datatype, in order to decide when to free it. Also, one
20
           may implement constructors of derived datatypes so that they keep pointers to their
21
           datatype arguments, rather then copying them. In this case, one needs to keep track
22
           of active datatype definition references in order to know when a datatype object can
23
           be freed. (End of advice to implementors.)
^{24}
25
     4.1.10 Duplicating a Datatype
26
27
28
     MPI_TYPE_DUP(oldtype, newtype)
29
       IN
                 oldtype
                                             datatype (handle)
30
31
       OUT
                 newtype
                                             copy of oldtype (handle)
32
33
     int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
34
     MPI_Type_dup(oldtype, newtype, ierror) BIND(C)
35
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
36
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
37
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
38
39
     MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
40
          INTEGER OLDTYPE, NEWTYPE, IERROR
41
42
          MPI_TYPE_DUP is a type constructor which duplicates the existing
     oldtype with associated key values. For each key value, the respective copy callback function
43
     determines the attribute value associated with this key in the new communicator; one
44
     particular action that a copy callback may take is to delete the attribute from the new
45
     datatype. Returns in newtype a new datatype with exactly the same properties as oldtype
46
```

and any copied cached information, see Section 6.7.4 on page 275. The new datatype has

identical upper bound and lower bound and yields the same net result when fully decoded

47

with the functions in Section 4.1.13. The newtype has the same committed state as the old oldtype.

#### 4.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI\_SEND(buf, count, datatype, ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI\_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

MPI\_TYPE\_CONTIGUOUS(count, datatype, newtype)
MPI\_TYPE\_COMMIT(newtype)
MPI\_SEND(buf, 1, newtype, dest, tag, comm)
MPI\_TYPE\_FREE(newtype).

Similar statements apply to all other communication functions that have a **count** and **datatype** argument.

Suppose that a send operation MPI\_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

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

and extent *extent*. (Explicit lower bound and upper bound markers are not listed in the type map, but they affect the value of *extent*.) The send operation sends  $n \cdot \text{count}$  entries, where entry  $i \cdot n + j$  is at location  $addr_{i,j} = \text{buf} + extent \cdot i + disp_j$  and has type  $type_j$ , for  $i = 0, \ldots, \text{count} - 1$  and  $j = 0, \ldots, n - 1$ . These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address  $addr_{i,j}$  in the calling program should be of a type that matches  $type_j$ , where type matching is defined as in Section 3.3.1. The message sent contains  $n \cdot \text{count}$  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), \ldots, (type_{n-1}, disp_{n-1})\},\$ 

with extent *extent*. (Again, explicit lower bound and upper bound markers 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  $\text{buf} + extent \cdot i + disp_j$  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.

 $^{24}$ 

```
1
      . . .
\mathbf{2}
     CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
3
      CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
4
      CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)
\mathbf{5}
      . . .
6
     CALL MPI_SEND(a, 4, MPI_REAL, ...)
7
     CALL MPI_SEND(a, 2, type2, ...)
8
     CALL MPI_SEND(a, 1, type22, ...)
9
      CALL MPI_SEND(a, 1, type4, ...)
10
      . . .
^{11}
      CALL MPI_RECV(a, 4, MPI_REAL, ...)
12
      CALL MPI_RECV(a, 2, type2, ...)
13
      CALL MPI_RECV(a, 1, type22, ...)
14
      CALL MPI_RECV(a, 1, type4, ...)
15
     Each of the sends matches any of the receives.
16
          A datatype may specify overlapping entries. The use of such a datatype in a receive
17
      operation is erroneous. (This is erroneous even if the actual message received is short enough
18
      not to write any entry more than once.)
19
          Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed,
20
      where datatype has type map,
21
22
           \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}.
23
^{24}
      The received message need not fill all the receive buffer, nor does it need to fill a number of
25
     locations which is a multiple of n. Any number, k, of basic elements can be received, where
26
     0 \le k \le \text{count} \cdot n. The number of basic elements received can be retrieved from status using
27
      the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.
28
29
      MPI_GET_ELEMENTS(status, datatype, count)
30
^{31}
       IN
                 status
                                              return status of receive operation (Status)
32
       IN
                                              datatype used by receive operation (handle)
                 datatype
33
       OUT
                 count
                                              number of received basic elements (integer)
34
35
      int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,
36
37
                     int *count)
38
     MPI_Get_elements(status, datatype, count, ierror) BIND(C)
39
          TYPE(MPI_Status), INTENT(IN) :: status
40
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                   datatype
41
          INTEGER, INTENT(OUT) :: count
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
43
     MPI_GET_ELEMENTS (STATUS, DATATYPE, COUNT, IERROR)
44
45
          INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
46
47
48
```

MPI\_GET\_ELEMENTS\_X(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

### 

- MPI\_Get\_elements\_x(status, datatype, count, ierror) BIND(C)
   TYPE(MPI\_Status), INTENT(IN) :: status
   TYPE(MPI\_Datatype), INTENT(IN) :: datatype
   INTEGER(KIND = MPI\_COUNT\_KIND), INTENT(OUT) :: count
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- MPI\_GET\_ELEMENTS\_X(STATUS, DATATYPE, COUNT, IERROR)
  INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, IERROR
  INTEGER(KIND=MPI\_COUNT\_KIND) COUNT

The datatype argument should match the argument provided by the receive call that set the status variable. For both functions, if the 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.

The previously defined function MPI\_GET\_COUNT (Section 3.2.5), has a different behavior. It returns the number of "top-level entries" received, i.e. the number of "copies" of type datatype. In the previous example, MPI\_GET\_COUNT may return any integer value k, where  $0 \le k \le \text{count.}$  If MPI\_GET\_COUNT returns k, then the number of basic elements received (and the value returned by MPI\_GET\_ELEMENTS or MPI\_GET\_ELEMENTS\_X) is  $n \cdot k$ . If the number of basic elements received is not a multiple of n, that is, if the receive operation has not received an integral number of datatype "copies," then MPI\_GET\_COUNT sets the value of count to MPI\_UNDEFINED.

**Example 4.12** Usage of MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS.

```
33
. . .
                                                                                   34
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
                                                                                   35
CALL MPI_TYPE_COMMIT(Type2, ierr)
                                                                                   36
. . .
                                                                                   37
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                   38
IF (rank.EQ.0) THEN
                                                                                   39
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
                                                                                   40
                                                                                   41
ELSE IF (rank.EQ.1) THEN
                                                                                   42
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                     ! returns i=1
                                                                                   43
                                                                                   44
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
                                                                                   45
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                                                   46
                                                    ! returns i=MPI_UNDEFINED
                                                                                   47
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3
                                                                                   48
END IF
```

 $^{24}$ 

<sup>1</sup> The functions MPI\_GET\_ELEMENTS and MPI\_GET\_ELEMENTS\_X can also be used <sup>2</sup> after a probe to find the number of elements in the probed message. Note that the <sup>3</sup> MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X return the same <sup>4</sup> values when they are used with basic datatypes as long as the limits of their respective <sup>5</sup> count arguments are not exceeded.

*Rationale.* The extension given to the definition of MPI\_GET\_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI\_GET\_ELEMENTS or MPI\_GET\_ELEMENTS\_X. (*End of rationale.*)

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified 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 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.*)

#### 24 25 26

27

28

29

30

 $^{31}$ 

32

33

34

35 36

37

38 39

40

41

42

43

6

7

8 9

10

11

12

13

14

15 16

17

18

19

20

21

22

23

### 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.

<sup>44</sup> A correct program uses only valid addresses to identify the locations of entries in <sup>46</sup> communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) <sup>47</sup> difference u - v can be computed only if both u and v are in the same sequential storage. <sup>48</sup> 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.*)

#### 4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

# MPI\_TYPE\_GET\_ENVELOPE(datatype, num\_integers, num\_addresses, num\_datatypes, combiner)

	combiner)		33	
IN	datatype	datatype to access (handle)	34	
OUT	num_integers	number of input integers used in the call constructing	35 36	
		combiner (non-negative integer)	37	
OUT	num_addresses	number of input addresses used in the call construct-	38	
		ing combiner (non-negative integer)	39	
OUT	num_datatypes	number of input datatypes used in the call construct-	40	
		ing combiner (non-negative integer)	41	
OUT	combiner	combiner (state)	42	
			43	
int MDT T	<pre>int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>			
INC MIT_I		int *num_datatypes, int *combiner)	45	
	int inum_addresses,	ind indm_dddddypeb, ind wedmbinei)	46	

 $\mathbf{2}$ 

 $^{24}$ 

1	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
2	<pre>INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,</pre>		
3	combiner		
4	INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	
5	MPT TYPE GET ENVELOPE (DATATYPE, NUM	L_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,	
6	COMBINER, IERROR)	,	
7	-	NUM_ADDRESSES, NUM_DATATYPES, COMBINER,	
8	IERROR	,,,,,,,,,,	
9			
10 11		ET_ENVELOPE returns information on the num-	
12		he call that created the datatype. The number-of-	
13	0	provide sufficiently large arrays in the decoding	
14		his call and the meaning of the returned values is	
15		e MPI datatype constructor call that was used in	
16	creating datatype.		
17	<i>Rationale</i> . By requiring that the co	mbiner reflect the constructor used in the creation	
18	· · ·	tion can be used to effectively recreate the calling	
19		a. This is the most useful information and was felt	
20		strains implementations to remember the original	
21	constructor sequence even if the inte		
22	The decoded information keeps trac	k of datatype duplications. This is important as	
23	-		
24	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		
25	derived datatype that can be freed. ( <i>End of rationale.</i> )		
26		(	
27	The list in Table $4.1$ has the values t	that can be returned in <b>combiner</b> on the left and	
28	the call associated with them on the right		
29			
30	MPI_COMBINER_NAMED	a named predefined datatype MPI_TYPE_DUP	
31 32	MPI_COMBINER_DUP MPI_COMBINER_CONTIGUOUS	MPI_TYPE_DOP MPI_TYPE_CONTIGUOUS	
33	MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	
34	MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	
35	MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	
36	MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	
37	MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	
38	MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	
39	MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	
40	MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	
41	MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	
42	MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	
43	MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX	
44	MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	
45	MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	
46			
47	Table 4.1. combiner values return	ned from MPI_TYPE_GET_ENVELOPE	
48	TABLE 4.1. COMDINEL VALUES LEUUI		

If combiner is MPI\_COMBINER\_NAMED then datatype is a named predefined datatype. The actual arguments used in the creation call for a datatype can be obtained using MPI\_TYPE\_GET\_CONTENTS.

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes,
array_of_integers, array_of_addresses, array_of_datatypes)

IN	datatype	datatype to access (handle)	8	
IN	max_integers	number of elements in array_of_integers (non-negative	9	
	5	integer)	10	
IN	max_addresses	number of elements in array_of_addresses (non-negative	11 12	
		integer)	12	
IN	max_datatypes	number of elements in array_of_datatypes (non-negative	14	
	max_uatatypes	integer)	15	
Ουτ	array of integers	- ,	16	
001	array_of_integers	contains integer arguments used in constructing datatype (array of integers)	17	
0.UT	C LL		18	
OUT	array_of_addresses	contains address arguments used in constructing	19	
		datatype (array of integers)	20	
OUT	array_of_datatypes	contains datatype arguments used in constructing	21	
		datatype (array of handles)	22 23	
			23	
int MPI_		atype datatype, int max_integers,	25	
		<pre>nt max_datatypes, int array_of_integers[],</pre>	26	
	MPI_Aint array_of_ad MPI_Datatype array_o		27	
		••	28	
MPI_Type		<pre>ax_integers, max_addresses, max_datatypes,</pre>	29	
		rray_of_addresses, array_of_datatypes,	30	
יועעיי	ierror) BIND(C)		31	
	(MPI_Datatype), INTENT(IN	) :: datatype ntegers, max_addresses, max_datatypes	32	
		y_of_integers(max_integers)	33 34	
	GER(KIND=MPI_ADDRESS_KIND		35	
	y_of_addresses(max_addres		36	
•		<pre>T) :: array_of_datatypes(max_datatypes)</pre>	37	
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	38	
MDT TVDF	CET CONTENTS (DATATYDE M	AX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	39	
		RRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	40	
	IERROR)		41	
INTE		S, MAX_ADDRESSES, MAX_DATATYPES,	42	
	Y_OF_INTEGERS(*), ARRAY_O		43	
INTE	GER(KIND=MPI_ADDRESS_KIND	) ARRAY_OF_ADDRESSES(*)	44	
datat	ne must be a predefined upp	amed or a derived datatype; the call is erroneous if	$45 \\ 46$	
-	datatype is a predefined named datatype.			

1

 $^{2}$ 

 $\overline{7}$ 

1	The values given for max_integers, max_addresses, and max_datatypes must be at least as
2	large as the value returned in num_integers, num_addresses, and num_datatypes, respectively,
3	in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.
4	
5	<i>Rationale.</i> The arguments max_integers, max_addresses, and max_datatypes allow for
6	error checking in the call. (End of rationale.)
7	
8	The datatypes returned in array_of_datatypes are handles to datatype objects that
9	are equivalent to the datatypes used in the original construction call. If these were derived
10	datatypes, then the returned datatypes are new datatype objects, and the user is responsible
11	for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then
12	the returned datatype is equal to that (constant) predefined datatype and cannot be freed.
12	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
14	undefined.
15	Note that MPI_TYPE_GET_CONTENTS can be invoked with a
16	datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL,
17	
18	MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed
19	predefined datatype). In such a case, an empty array_of_datatypes is returned.
20	Rationale. The definition of datatype equivalence implies that equivalent predefined
21	
22	datatypes are equal. By requiring the same handle for named predefined datatypes,
23	it is possible to use the == or .EQ. comparison operator to determine the datatype
24	involved. (End of rationale.)
25	Advise to implementation. The detector of the summer of detector and summer a
26	Advice to implementors. The datatypes returned in array_of_datatypes must appear
27	to the user as if each is an equivalent copy of the datatype used in the type constructor
28	call. Whether this is done by creating a new datatype or via another mechanism such
29	as a reference count mechanism is up to the implementation as long as the semantics
30	are preserved. (End of advice to implementors.)
31	
32	Rationale. The committed state and attributes of the returned datatype is delib-
33	erately left vague. The datatype used in the original construction may have been
34	modified since its use in the constructor call. Attributes can be added, removed, or
35	modified as well as having the datatype committed. The semantics given allow for
36	a reference count implementation without having to track these changes. (End of
37	rationale.)
38	
39	In the deprecated datatype constructor calls, the address arguments in Fortran are
39 40	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 ad-
41	dresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the
42	deprecated calls were used. Thus, the location of values returned can be thought of as being
43	returned by the C bindings. It can also be determined by examining the preferred calls for
44	datatype constructors for the deprecated calls that involve addresses.
45	
46	Rationale. By having all address arguments returned in the
47	<code>array_of_addresses</code> argument, the result from a C and Fortran decoding of a <code>datatype</code>
48	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.*)

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:

PARAMETER (LARGE = 1000) 10 INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 11 INTEGER (KIND=MPI\_ADDRESS\_KIND) A(LARGE) 12! CONSTRUCT DATATYPE TYPE (NOT SHOWN) 13 CALL MPI\_TYPE\_GET\_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR) 14 IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN 15WRITE (\*, \*) "NI, NA, OR ND = ", NI, NA, ND, & 16" RETURNED BY MPI\_TYPE\_GET\_ENVELOPE IS LARGER THAN LARGE = ", LARGE 17CALL MPI\_ABORT(MPI\_COMM\_WORLD, 99, IERROR) 18 ENDIF 19 CALL MPI\_TYPE\_GET\_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR) 20

or in C the analogous calls of:

```
23
#define LARGE 1000
                                                                                        ^{24}
int ni, na, nd, combiner, i[LARGE];
                                                                                        25
MPI_Aint a[LARGE];
                                                                                        26
MPI_Datatype type, d[LARGE];
                                                                                        27
/* construct datatype type (not shown) */
                                                                                        28
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
                                                                                        29
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
                                                                                        30
    fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
                                                                                        31
    fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
                                                                                        32
             LARGE);
                                                                                        33
    MPI_Abort(MPI_COMM_WORLD, 99);
                                                                                        34
};
                                                                                        35
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
                                                                                        36
                                                                                        37
    In the descriptions that follow, the lower case name of arguments is used.
                                                                                        38
    If combiner is MPI_COMBINER_NAMED then it is erroneous to call
                                                                                        39
MPI_TYPE_GET_CONTENTS.
                                                                                        40
    If combiner is MPI_COMBINER_DUP then
                                                                                        41
                              Fortran location
 Constructor argument
                         С
                                                                                        42
 oldtype
                        d[0]
                                   D(1)
                                                                                        43
and ni = 0, na = 0, nd = 1.
                                                                                        44
    If combiner is MPI_COMBINER_CONTIGUOUS then
                                                                                        45
 Constructor argument
                              Fortran location
                         С
                                                                                        46
 count
                        i[0]
                                   I(1)
                                                                                        47
                        d[0]
                                   D(1)
 oldtype
                                                                                        48
and ni = 1, na = 0, nd = 1.
```

1 2

3

4

5

6

7

8 9

21

If combiner is MPI_C	OMBINER VECTO	R then
Constructor argument	C Fortran loo	
count	i[0] I(1)	
blocklength	i[1] $I(2)$	
stride	i[2] $I(3)$	
oldtype	d[0] $D(1)$	
$\overline{\text{and ni} = 3, \text{ na} = 0, \text{ nd}} =$		
If combiner is MPI_C		OR then
Constructor argument	C Fortran loo	
count	i[0] I(1)	
blocklength	i[1] I(2)	
stride	a[0]   A(1)	
oldtype	d[0]  D(1)	
and $ni = 2$ , $na = 1$ , $nd =$	= 1.	
If combiner is MPI_C	OMBINER_INDEXE	
Constructor argument	С	Fortran location
count	i[0]	I(1)
$array_of_blocklengths$	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
array_of_displacements	i[i[0]+1] to $i[2*i$	[0]] I(I(1)+2) to I(2*I(1)+2)
oldtype	d[0]	D(1)
and $ni = 2*count+1$ , na	$= 0,  \mathrm{nd} = 1.$	
If combiner is MPI_C	OMBINER_HINDE>	
Constructor argument	С	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
array_of_displacements	a[0] to $a[i[0]-1]$	A(1) to $A(I(1))$
oldtype	d[0]	D(1)
and $ni = count+1$ , $na =$		
If combiner is MPI_C		
Constructor argument	C	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements		
oldtype	d[0]	D(1)
and $ni = count+2$ , $na =$		
If combiner is MPI_C	<u></u>	
Constructor argument	<u> </u>	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements		A(I) = A(I(I))
1.1.		A(1) to $A(I(1))$
oldtype	d[0]	$\begin{array}{c} A(1) \text{ to } A(1(1)) \\ D(1) \end{array}$
and $ni = 2$ , $na = count$ ,	$\frac{d[0]}{nd = 1.}$	D(1)
and $ni = 2$ , $na = count$ , If combiner is MPI_C	$\frac{d[0]}{nd = 1.}$	D(1) T then
and ni = 2, na = count, If combiner is MPI_C Constructor argument	$\frac{d[0]}{nd = 1.}$	D(1) <u>T then</u> Fortran location
and $ni = 2$ , $na = count$ , If combiner is MPI_C Constructor argument count	$\frac{d[0]}{\text{nd} = 1.}$ $\frac{\text{COMBINER_STRUC}}{C}$	D(1)       T then       Fortran location       I(1)
and ni = 2, na = count, If combiner is MPI_C Constructor argument count array_of_blocklengths	$\frac{d[0]}{\text{nd} = 1.}$ $\frac{\text{COMBINER_STRUC}}{C}$ $i[0]$ $i[1] \text{ to } i[i[0]]$	$     \begin{array}{r} D(1) \\ \hline D(1) \\ \hline D(1) \\ \hline I(1) \\ I(2) \text{ to } I(I(1)+1) \end{array} $
and $ni = 2$ , $na = count$ , If combiner is MPI_C Constructor argument count	$\frac{d[0]}{\text{nd} = 1.}$ $\frac{\text{COMBINER_STRUC}}{C}$ $i[0]$ $i[1] \text{ to } i[i[0]]$	$     \begin{array}{r} D(1) \\ \hline D(1) \\ \hline D(1) \\ \hline I(1) \\ I(2) \text{ to } I(I(1)+1) \end{array} $

Constructor argument	COMBINER_SUBARRAY	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
array_of_subsizes		I(I(1)+2) to $I(2*I(1)+1)$
array_of_starts		$I(2^*I(1)+2) \text{ to } I(2^*I(1)+1)$ $I(2^*I(1)+2) \text{ to } I(3^*I(1)+1)$
order	i[3*i[0]+1]	$I(2 I(1)+2) \approx I(0 I(1)+1)$ I(3*I(1)+2]
oldtype	$\frac{d[0]}{d[0]}$	D(1)
and ni = $3$ *ndims+2, na	L J	_ (-)
	COMBINER_DARRAY th	${ m en}$
Constructor argument	C	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to $i[i[2]+2]$	I(4)  to  I(I(3)+3)
array_of_distribs	i[i[2]+3] to $i[2*i[2]+$	2] $I(I(3)+4)$ to $I(2*I(3)+3)$
array_of_dargs		+2] $I(2*I(3)+4)$ to $I(3*I(3)+3)$
array_of_psizes	i[3*i[2]+3] to $i[4*i[2]-3]$	+2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	$\mathrm{D}(1)$
and $ni = 4*ndims + 4$ , na	a = 0, nd = 1.	
If combiner is MPI_0	COMBINER_F90_REAL t	
Constructor argument	C Fortran location	n
р	i[0] I(1)	
r	i[1] I(2)	
and $ni = 2$ , $na = 0$ , $nd =$		
	COMBINER_F90_COMPL	
Constructor argument	C Fortran location	<u>n</u>
р	i[0] $I(1)$	
r	i[1] I(2)	
and $ni = 2$ , $na = 0$ , $nd =$		
Constructor argument	C Fortran location	
	i[0] $I(1)$	<u> </u>
$\frac{\mathbf{r}}{\mathbf{and}\ \mathbf{ni} = 1,\ \mathbf{na} = 0,\ \mathbf{nd} = 1}$		
	– 0. COMBINER_RESIZED th	en
Constructor argument	C Fortran locatio	
lb	a[0] A(1)	
	$\begin{array}{c} \mathbf{a}[0] \\ \mathbf{a}[1] \\ \mathbf{A}(2) \end{array}$	
extent		
extent oldtype	$\begin{array}{c} \mathbf{a}[1] \\ \mathbf{d}[0] \\ \end{array} \qquad \mathbf{D}(1) \end{array}$	

### 4.1.14 Examples

The following examples illustrate the use of derived datatypes.	45
	46
<b>Example 4.13</b> Send and receive a section of a 3D array.	47

```
1
           REAL a(100,100,100), e(9,9,9)
\mathbf{2}
           INTEGER oneslice, twoslice, threeslice, myrank, ierr
3
           INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
4
           INTEGER status(MPI_STATUS_SIZE)
5
6
     С
           extract the section a(1:17:2, 3:11, 2:10)
7
     С
           and store it in e(:,:,:).
8
9
           CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
10
11
           CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
12
13
     С
           create datatype for a 1D section
14
           CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr)
15
16
     С
           create datatype for a 2D section
17
           CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*sizeofreal, oneslice,
18
                                          twoslice, ierr)
19
20
     С
           create datatype for the entire section
21
           CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice,
22
                                          threeslice, ierr)
23
^{24}
           CALL MPI_TYPE_COMMIT(threeslice, ierr)
25
           CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
26
                               MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
27
28
     Example 4.14 Copy the (strictly) lower triangular part of a matrix.
29
30
           REAL a(100,100), b(100,100)
^{31}
           INTEGER disp(100), blocklen(100), ltype, myrank, ierr
32
           INTEGER status(MPI_STATUS_SIZE)
33
34
     С
           copy lower triangular part of array a
35
     С
           onto lower triangular part of array b
36
37
           CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
38
39
     С
           compute start and size of each column
40
           DO i=1, 100
41
             disp(i) = 100*(i-1) + i
42
             blocklen(i) = 100-i
43
           END DO
44
45
           create datatype for lower triangular part
     С
46
           CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)
47
48
           CALL MPI_TYPE_COMMIT(ltype, ierr)
```

CALL MPI_SENDRECV(a, 1, lt	ype, myrank	x, 0, b, 1,		
ltype, m	yrank, O, M	IPI_COMM_WORLD,	status,	ierr)

Example 4.15 Transpose a matrix.

6 REAL a(100,100), b(100,100) 7 INTEGER row, xpose, myrank, ierr 8 INTEGER (KIND=MPI\_ADDRESS\_KIND) lb, sizeofreal 9 INTEGER status(MPI\_STATUS\_SIZE) 10 11 С transpose matrix a onto b 1213 CALL MPI\_COMM\_RANK(MPI\_COMM\_WORLD, myrank, ierr) 1415CALL MPI\_TYPE\_GET\_EXTENT(MPI\_REAL, lb, sizeofreal, ierr) 1617С create datatype for one row 18 CALL MPI\_TYPE\_VECTOR(100, 1, 100, MPI\_REAL, row, ierr) 1920С create datatype for matrix in row-major order 21CALL MPI\_TYPE\_CREATE\_HVECTOR(100, 1, sizeofreal, row, xpose, ierr) 22 23CALL MPI\_TYPE\_COMMIT(xpose, ierr)  $^{24}$ 25С send matrix in row-major order and receive in column major order 26CALL MPI\_SENDRECV(a, 1, xpose, myrank, 0, b, 100\*100, 27MPI\_REAL, myrank, 0, MPI\_COMM\_WORLD, status, ierr) 28 29 30 **Example 4.16** Another approach to the transpose problem: 31REAL a(100,100), b(100,100) 32 INTEGER row, row1 33 INTEGER (KIND=MPI\_ADDRESS\_KIND) disp(2), lb, sizeofreal 34 INTEGER myrank, ierr 35 INTEGER status(MPI\_STATUS\_SIZE) 36 37 CALL MPI\_COMM\_RANK(MPI\_COMM\_WORLD, myrank, ierr) 38 39 С transpose matrix a onto b 4041 CALL MPI\_TYPE\_GET\_EXTENT(MPI\_REAL, lb, sizeofreal, ierr) 4243 С create datatype for one row 44CALL MPI\_TYPE\_VECTOR(100, 1, 100, MPI\_REAL, row, ierr) 4546С create datatype for one row, with the extent of one real number 471b = 048

```
1
           CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)
\mathbf{2}
3
           CALL MPI_TYPE_COMMIT(row1, ierr)
4
5
           send 100 rows and receive in column major order
     С
6
           CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100,
\overline{7}
                               MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
8
9
     Example 4.17 We manipulate an array of structures.
10
11
     struct Partstruct
12
     ſ
13
                type; /* particle type */
        int
14
        double d[6]; /* particle coordinates */
15
                      /* some additional information */
        char
               b[7];
16
     };
17
18
                           particle[1000];
     struct Partstruct
19
20
     int
                   i, dest, tag;
21
     MPI_Comm
                   comm;
22
23
^{24}
     /* build datatype describing structure */
25
26
     MPI_Datatype Particlestruct, Particletype;
27
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
28
     int
                   blocklen[3] = \{1, 6, 7\};
29
     MPI_Aint
                   disp[3];
30
     MPI_Aint
                   base, lb, sizeofentry;
^{31}
32
33
     /* compute displacements of structure components */
34
35
     MPI_Get_address(particle, disp);
36
     MPI_Get_address(particle[0].d, disp+1);
37
     MPI_Get_address(particle[0].b, disp+2);
38
     base = disp[0];
39
     for (i=0; i < 3; i++) disp[i] -= base;</pre>
40
41
     MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);
42
43
        /* If compiler does padding in mysterious ways,
44
        the following may be safer */
45
46
     /* compute extent of the structure */
47
48
     MPI_Get_address(particle+1, &sizeofentry);
```

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

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

**Example 4.18** The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
                                                                                       4
{
                                                                                       5
    int
            type;
                                                                                       6
    double d[6];
                                                                                       7
    char
            b[7];
                                                                                       8
};
                                                                                       9
                                                                                       10
struct Partstruct particle[1000];
                                                                                       11
                                                                                       12
            /* build datatype describing first array entry */
                                                                                       13
                                                                                       14
MPI_Datatype Particletype;
                                                                                       15
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
                                                                                       16
              block[3] = \{1, 6, 7\};
int
                                                                                       17
MPI_Aint
              disp[3];
                                                                                       18
                                                                                       19
MPI_Get_address(particle, disp);
                                                                                       20
MPI_Get_address(particle[0].d, disp+1);
                                                                                       21
MPI_Get_address(particle[0].b, disp+2);
                                                                                       22
MPI_Type_create_struct(3, block, disp, type, &Particletype);
                                                                                       23
                                                                                       24
/* Particletype describes first array entry -- using absolute
                                                                                       25
   addresses */
                                                                                       26
                                                                                       27
                   /* 5.1:
                                                                                       28
             send the entire array */
                                                                                       29
                                                                                       30
MPI_Type_commit(&Particletype);
                                                                                       ^{31}
MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
                                                                                       32
                                                                                       33
                                                                                       34
                  /* 5.2:
                                                                                       35
         send the entries of type zero,
                                                                                       36
         preceded by the number of such entries */
                                                                                       37
                                                                                       38
MPI_Datatype Zparticles, Ztype;
                                                                                       39
                                                                                       40
int
              zdisp[1000];
                                                                                       41
int
              zblock[1000], i, j, k;
                                                                                       42
              zzblock[2] = {1,1};
int
                                                                                       43
MPI_Datatype zztype[2];
                                                                                       44
MPI_Aint
              zzdisp[2];
                                                                                       45
                                                                                       46
j=0;
                                                                                       47
for (i=0; i < 1000; i++)</pre>
                                                                                       48
```

1

 $\mathbf{2}$ 

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

```
1
MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
                                                                                      2
                                                                                      3
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
                                                                                      4
/* actual communication */
                                                                                      5
                                                                                      6
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
                                                                                      7
                                                                                      8
                                                                                      9
Example 4.20 This example shows how a datatype can be decoded. The routine
                                                                                      10
printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
                                                                                      11
datatypes that are not predefined.
                                                                                      12
                                                                                      13
/*
                                                                                      14
  Example of decoding a datatype.
                                                                                      15
                                                                                      16
  Returns 0 if the datatype is predefined, 1 otherwise
                                                                                      17
 */
                                                                                      18
#include <stdio.h>
                                                                                      19
#include <stdlib.h>
#include "mpi.h"
                                                                                      20
                                                                                     21
int printdatatype(MPI_Datatype datatype)
                                                                                     22
{
                                                                                     23
    int *array_of_ints;
                                                                                      24
    MPI_Aint *array_of_adds;
                                                                                     25
    MPI_Datatype *array_of_dtypes;
                                                                                      26
    int num_ints, num_adds, num_dtypes, combiner;
                                                                                     27
    int i;
                                                                                     28
                                                                                     29
    MPI_Type_get_envelope(datatype,
                            &num_ints, &num_adds, &num_dtypes, &combiner);
                                                                                     30
                                                                                      31
    switch (combiner) {
                                                                                     32
    case MPI_COMBINER_NAMED:
                                                                                     33
        printf("Datatype is named:");
                                                                                     34
        /* To print the specific type, we can match against the
            predefined forms. We can NOT use a switch statement here
                                                                                     35
                                                                                     36
            We could also use MPI_TYPE_GET_NAME if we prefered to use
                                                                                     37
            names that the user may have changed.
                                                                                     38
          */
                                                                                     39
                                            printf( "MPI_INT\n" );
        if
                 (datatype == MPI_INT)
        else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
                                                                                      40
                                                                                      41
         ... else test for other types ...
                                                                                      42
        return 0;
        break;
                                                                                      43
                                                                                      44
    case MPI_COMBINER_STRUCT:
                                                                                      45
    case MPI_COMBINER_STRUCT_INTEGER:
                                                                                      46
        printf("Datatype is struct containing");
                                                                                      47
                        = (int *)malloc(num_ints * sizeof(int));
        array_of_ints
                                                                                      48
        array_of_adds
                          =
```

1	<pre>(MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));</pre>
2	array_of_dtypes = (MPI_Datatype *)
3	<pre>malloc(num_dtypes * sizeof(MPI_Datatype));</pre>
4	MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
5	array_of_ints, array_of_adds, array_of_dtypes);
6	<pre>printf(" %d datatypes:\n", array_of_ints[0]);</pre>
7	for (i=0; i <array_of_ints[0]; i++)="" td="" {<=""></array_of_ints[0];>
8	printf("blocklength %d, displacement %ld, type:\n",
9	array_of_ints[i+1], (long)array_of_adds[i]);
10	<pre>if (printdatatype(array_of_dtypes[i])) {</pre>
11	/* Note that we free the type ONLY if it
12	is not predefined */
13	<pre>MPI_Type_free(&amp;array_of_dtypes[i]);</pre>
14	}
15	}
16	<pre>free(array_of_ints);</pre>
17	<pre>free(array_of_adds);</pre>
18	<pre>free(array_of_dtypes);</pre>
19	break;
20	other combiner values
21	default:
22	<pre>printf("Unrecognized combiner type\n");</pre>
23	}
24	return 1;
25	}
26	

## 4.2 Pack and Unpack

29Some existing communication libraries provide pack/unpack functions for sending noncon-30 tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending  $^{31}$ it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are 32 described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. 33 The user specifies the layout of the data to be sent or received, and the communication 34library directly accesses a noncontiguous buffer. The pack/unpack routines are provided 35 for compatibility with previous libraries. Also, they provide some functionality that is not 36 otherwise available in MPI. For instance, a message can be received in several parts, where 37 the receive operation done on a later part may depend on the content of a former part. 38 Another use is that outgoing messages may be explicitly buffered in user supplied space, 39 thus overriding the system buffering policy. Finally, the availability of pack and unpack 40 operations facilitates the development of additional communication libraries layered on top  $^{41}$ of MPI.

27

28

43

44

 $45 \\ 46$ 

47

MPI_PACK	(inbuf, incount, datatype, outb	ouf, outsize, position, comm)	1
IN	inbuf	input buffer start (choice)	2
		•	3
IN	incount	number of input data items (non-negative integer)	4
IN	datatype	datatype of each input data item (handle)	5
OUT	outbuf	output buffer start (choice)	6 7
IN	outsize	output buffer size, in bytes (non-negative integer)	8
INOUT	position	current position in buffer, in bytes (integer)	9
IN	comm	communicator for packed message (handle)	10
			11
int MPI P	ack(const void* inbuf. in	nt incount, MPI_Datatype datatype,	12
		tsize, int *position, MPI_Comm comm)	13 14
		-	14
MPI_Pack(	BIND(C)	, outbuf, outsize, position, comm, ierror)	16
TVDF (	*), DIMENSION(), INTEN	$\Gamma(TN)$ · · · inhuf	17
	*), DIMENSION(), INTEN: *), DIMENSION() :: ou		18
	ER, INTENT(IN) :: incour		19
	MPI_Datatype), INTENT(IN)	-	20
	ER, INTENT(INOUT) :: pos	0 I	21
	MPI_Comm), INTENT(IN) ::		22
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	23
MDT DACK	INDUE INCOUNT DATATYDE	, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)	24
_	<pre>&gt; INBUF(*), OUTBUF(*)</pre>	, UVIDUR, UVIDIZE, PUDITIUN, CUMMI, IERRUR)	25
01		ISIZE, POSITION, COMM, IERROR	26
			27
Packs	the message in the send buffer	specified by inbuf, incount, datatype into the buffer	28

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

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.

 $^{31}$ 

```
1
     MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
2
       IN
                 inbuf
                                              input buffer start (choice)
3
       IN
                 insize
                                              size of input buffer, in bytes (non-negative integer)
4
5
       INOUT
                 position
                                              current position in bytes (integer)
6
       OUT
                 outbuf
                                              output buffer start (choice)
7
       IN
                 outcount
                                              number of items to be unpacked (integer)
8
9
       IN
                 datatype
                                              datatype of each output data item (handle)
10
       IN
                 comm
                                              communicator for packed message (handle)
11
12
     int MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf,
13
                     int outcount, MPI_Datatype datatype, MPI_Comm comm)
14
15
     MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
16
                     ierror) BIND(C)
17
          TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
18
          TYPE(*), DIMENSION(..) :: outbuf
19
          INTEGER, INTENT(IN) :: insize, outcount
20
          INTEGER, INTENT(INOUT) :: position
21
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                  datatype
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
     MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
25
                     IERROR)
26
          <type> INBUF(*), OUTBUF(*)
27
          INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
28
29
          Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from
30
     the buffer space specified by inbuf and insize. The output buffer can be any communication
31
     buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize
32
     bytes, starting at address inbuf. The input value of position is the first location in the input
33
     buffer occupied by the packed message. position is incremented by the size of the packed
34
     message, so that the output value of position is the first location in the input buffer after
35
     the locations occupied by the message that was unpacked. comm is the communicator used
36
     to receive the packed message.
37
38
                              Note the difference between MPI_RECV and MPI_UNPACK: in
```

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.)

39

40

41

42

43

44

45

46

47

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 and comm.

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.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

```
1
     MPI_PACK_SIZE(incount, datatype, comm, size)
\mathbf{2}
       IN
                 incount
                                             count argument to packing call (non-negative integer)
3
       IN
                                             datatype argument to packing call (handle)
                 datatype
4
5
       IN
                 comm
                                             communicator argument to packing call (handle)
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)
11
12
     MPI_Pack_size(incount, datatype, comm, size, ierror) BIND(C)
13
          INTEGER, INTENT(IN) :: incount
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
          A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound
22
     on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype,
23
     outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed
24
     by the size parameter, then MPI_PACK_SIZE sets the value of size to MPI_UNDEFINED.
25
           Rationale. The call returns an upper bound, rather than an exact bound, since the
26
           exact amount of space needed to pack the message may depend on the context (e.g.,
27
           first message packed in a packing unit may take more space). (End of rationale.)
28
29
30
     Example 4.21 An example using MPI_PACK.
^{31}
     int
                  position, i, j, a[2];
32
                  buff[1000];
     char
33
34
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
35
     if (myrank == 0)
36
     {
37
          /* SENDER CODE */
38
39
          position = 0;
40
          MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
41
          MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
42
          MPI_Send(buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
43
     }
44
     else /* RECEIVER CODE */
45
          MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
46
47
48
     Example 4.22 An elaborate example.
```

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

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

```
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] = ' \ ';
}
```

### 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 13.7.2, 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

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_PACk	<pre>K_EXTERNAL(datare</pre>	p, inbuf, incount, datatype, outbuf, outsize, position)		
2 3	IN	datarep	data representation (string)		
4	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 8	OUT	outbuf	output buffer start (choice)		
9	IN	outsize	output buffer size, in bytes (integer)		
10 11	INOUT	position	current position in buffer, in bytes (integer)		
12 13 14 15	int MPI_F		st char datarep[], const void *inbuf, int incount, datatype, void *outbuf, MPI_Aint outsize, sition)		
16	MPI_Pack_	external(datarep	, inbuf, incount, datatype, outbuf, outsize,		
17		-	rror) BIND(C)		
18			ENT(IN) :: datarep		
19 20		<pre>(*), DIMENSION() (*), DIMENSION()</pre>			
21		GER, INTENT(IN) ::			
22	TYPE	(MPI_Datatype), IN	NTENT(IN) :: datatype		
23	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize				
24	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
25 26	INTEC	ER, UPIIUNAL, IN	LENI(UUI) :: lerror		
27	MPI_PACK_		, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,		
28	T N T T C	POSITION, IE ER INCOUNT, DATAT			
29			ESS_KIND) OUTSIZE, POSITION		
30 31		ACTER*(*) DATAREP			
32	<type< td=""><td><pre>&gt; INBUF(*), OUTBU</pre></td><td>JF(*)</td></type<>	<pre>&gt; INBUF(*), OUTBU</pre>	JF(*)		
33					
34 35	MPI_UNP/	ACK_EXTERNAL(dat	tarep, inbuf, insize, position, outbuf, outsize, position)		
36	IN	datarep	data representation (string)		
37 38	IN inbut input buffer start (choice)		input buffer start (choice)		
39	IN insize input buffer size in bytes (integer)		input buffer size, in bytes (integer)		
40	INOUT	<b>PUT</b> position current position in buffer, in bytes (integer)			
41	OUT	outbuf	output buffer start (choice)		
42 43	IN	outcount	number of output data items (integer)		
44 45	IN	datatype	datatype of output data item (handle)		
45 46 47 48	int MPI_U	MPI_Aint ins	onst char datarep[], const void *inbuf, ize, MPI_Aint *position, void *outbuf, , MPI_Datatype datatype)		

MPI_Unpac	-	, insize, position, outbuf, outcount,	1
	datatype, ierror) BII		2
	CTER(LEN=*), INTENT(IN) :	-	3
	*), DIMENSION(), INTENT		4
	*), DIMENSION() :: ou		5 6
	ER(KIND=MPI_ADDRESS_KIND)	, INTENT(IN) :: INSIZE , INTENT(INOUT) :: position	7
	ER, INTENT(IN) :: outcou	-	8
	MPI_Datatype), INTENT(IN)		9
	ER, OPTIONAL, INTENT(OUT)		10
			11
MPI_UNPAC		, INSIZE, POSITION, OUTBUF, OUTCOUNT,	12
ΤΜͲΕΟ	DATATYPE, IERROR) ER OUTCOUNT, DATATYPE, IE	סחסס	13
	ER (KIND=MPI_ADDRESS_KIND)		14
	CTER*(*) DATAREP	INDIZE, I ODITION	15
	> INBUF(*), OUTBUF(*)		16
			17 18
			19
MPI_PACK	_EXTERNAL_SIZE(datarep, ir	icount, datatype, size)	20
IN	datarep	data representation (string)	21
		- ( 0)	22
IN	incount	number of input data items (integer)	23
IN	datatype	datatype of each input data item (handle)	24
OUT	size	output buffer size, in bytes (integer)	25
			26
int MPI_P	ack_external_size(const c	har datarep[], int incount,	27
	MPI_Datatype datatype	e, MPI_Aint *size)	28 29
MPT Pack	external size(datarep, in	count, datatype, size, ierror) BIND(C)	30
	MPI_Datatype), INTENT(IN)		31
	ER, INTENT(IN) :: incoun	• -	32
	CTER(LEN=*), INTENT(IN) :		33
INTEG	ER(KIND=MPI_ADDRESS_KIND)	, INTENT(OUT) :: size	34
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	35
MPT PACK	EXTERNAL STZE (DATABED IN	COUNT, DATATYPE, SIZE, IERROR)	36
	ER INCOUNT, DATATYPE, IER		37
	ER(KIND=MPI_ADDRESS_KIND)		38
CHARA	CTER*(*) DATAREP		39
			40 41
			41
			43
			44
			45
			46

# 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: 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).

4

• 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).

One of the key arguments in a call to a collective routine is a communicator that 5defines 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 7 operations are defined to be consistent with the syntax and semantics of the point-to-point 8 operations. Thus, general datatypes are allowed and must match between sending and re-9 ceiving 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 corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 4.1) between sender and receiver are still allowed.

Collective operations can (but are not required to) complete as soon as the caller's 21participation in the collective communication is finished. A blocking operation is complete 22 as soon as the call returns. A nonblocking (immediate) call requires a separate completion 23call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free  $^{24}$ to modify locations in the communication buffer. It does not indicate that other processes 25in the group have completed or even started the operation (unless otherwise implied by the 26description of the operation). Thus, a collective communication operation may, or may not, 27have the effect of synchronizing all calling processes. This statement excludes, of course, 28the barrier operation. 29

Collective communication calls may use the same communicators as point-to-point 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.

35 36

37

38

39

*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.

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

<sup>42</sup> (End of rationale.)

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

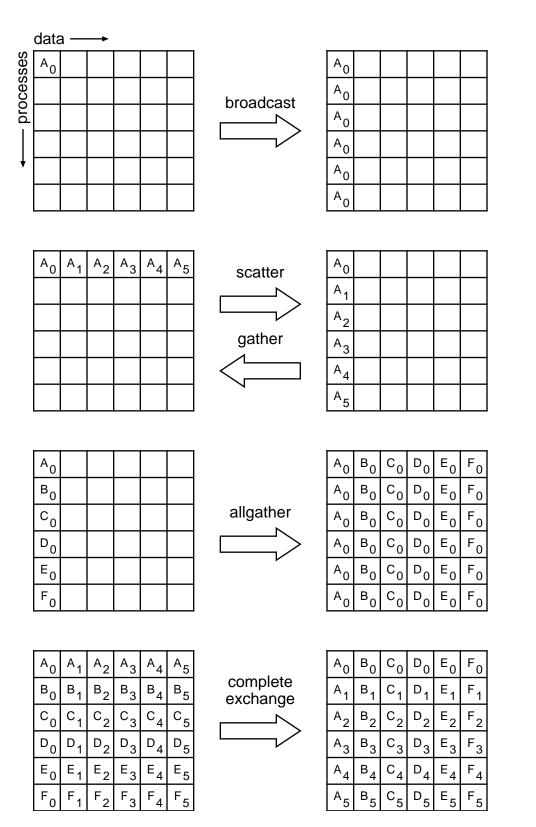


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data  $A_0$ , but after the broadcast all processes contain it.

 $\mathbf{2}$ 

 $^{31}$ 

 $44 \\ 45$ 

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.*)

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.

18 19

20

## 5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 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 can be thought of as an identifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

- 28
- 29 30

#### 5.2.1 Specifics for Intracommunicator Collective Operations

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

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.

37 38

39

40

41

42

43

44

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

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.

1

2

3

4

5 6

7 8

9

10

11

Note that MPI\_IN\_PLACE is a special kind of value; it has the same restrictions on its use that MPI\_BOTTOM has. (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.

<ul> <li>MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV</li> </ul>
• MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW
<ul> <li>MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER</li> </ul>

• 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, MPI\_EXSCAN, and MPI\_IEXSCAN do not fit this taxonomy.

The application of collective communication to intercommunicators is best described 35in terms of two groups. For example, an all-to-all MPI\_ALLGATHER operation can be 36 described as collecting data from all members of one group with the result appearing in all 37 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 39 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:

- MPI\_BARRIER, MPI\_IBARRIER
- MPI\_BCAST, MPI\_IBCAST

1 2

3 4

5

6

7

8 9

19 20

21

2223

 $^{24}$ 

2526

27

28 29

30  $^{31}$ 

32

33

34

38

40

41

42

43

44

4546

47

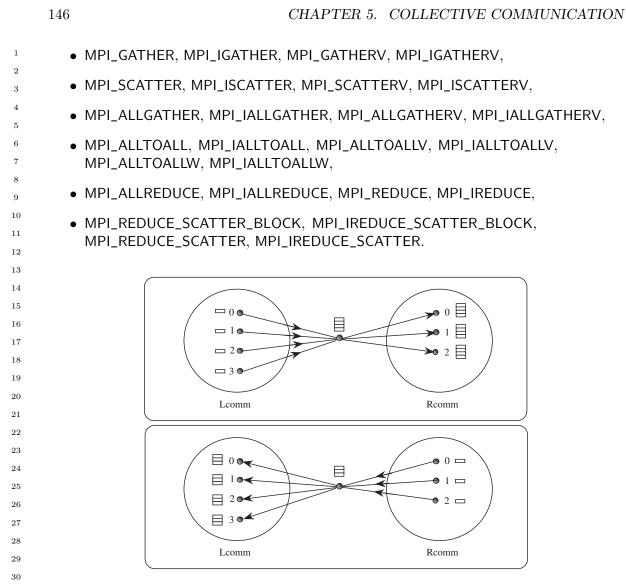


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

 $^{31}$ 

32

#### 5.2.3 Specifics for Intercommunicator Collective Operations

 $^{36}_{37}$  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.

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

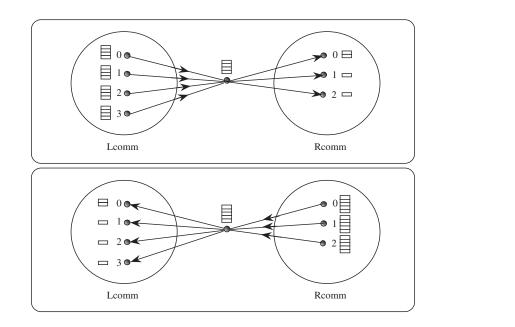


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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.*)

### 5.3 Barrier Synchronization

If comm is an intracommunicator, MPI\_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.  $44 \\ 45$ 

If comm is an intercommunicator, MPI\_BARRIER involves two groups. The call returns at processes in one group (group A) of the intercommunicator only after all members of the other group (group B) have entered the call (and vice versa). A process may return from the call before all processes in its own group have entered the call.

#### Broadcast 5.4

MPI\_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

int MPI\_Bcast(void\* buffer, int count, MPI\_Datatype datatype, int root, MPI\_Comm comm)

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror) BIND(C)
```

TYPE(\*), DIMENSION(..) :: buffer INTEGER, INTENT(IN) :: count, root

TYPE(MPI\_Datatype), INTENT(IN) :: datatype

TYPE(MPI\_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI\_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) 2829<type> BUFFER(\*) 30

INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

31If comm is an intracommunicator, MPI\_BCAST broadcasts a message from the process 32 with rank root to all processes of the group, itself included. It is called by all members of 33 the group using the same arguments for comm and root. On return, the content of root's 34 buffer is copied to all other processes. 35

General, derived datatypes are allowed for datatype. The type signature of count, 36 datatype on any process must be equal to the type signature of count, datatype at the root. 37 This implies that the amount of data sent must be equal to the amount received, pairwise 38 between each process and the root. MPI\_BCAST and all other data-movement collective 39 routines make this restriction. Distinct type maps between sender and receiver are still 40 allowed. 41

The "in place" option is not meaningful here.

42If comm is an intercommunicator, then the call involves all processes in the intercom-43 municator, but with one group (group A) defining the root process. All processes in the 44other group (group B) pass the same value in argument root, which is the rank of the root 45in group A. The root passes the value MPI\_ROOT in root. All other processes in group A 46 pass the value MPI\_PROC\_NULL in root. Data is broadcast from the root to all processes 47

1

 $\mathbf{2}$ 

3

4

56

2021

22

23

24

25

26

27

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.

### 5.5 Gather

MPI\_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)	25
IN	sendcount	number of elements in send buffer (non-negative inte-	26
		$\operatorname{ger})$	27
IN	sendtype	data type of send buffer elements (handle)	28
			29
OUT	recvbuf	address of receive buffer (choice, significant only at	30
		root)	31
IN	recvcount	number of elements for any single receive (non-negative	32
		integer, significant only at root)	33
IN	recvtype	data type of recv buffer elements (significant only at	34
		root) (handle)	35
			36
IN	root	rank of receiving process (integer)	37
IN	comm	communicator (handle)	38
		· · ·	39

int MPI\_Gather(const void\* sendbuf, int sendcount, MPI\_Datatype sendtype, void\* recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm)

 $^{24}$ 

1	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2	TYPE(MPI_Comm), INTENT(IN) :: comm
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	
5	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
6	ROOT, COMM, IERROR)
7	<type> SENDBUF(*), RECVBUF(*)</type>
8	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
9	If comm is an intracommunicator, each process (root process included) sends the con-
10	tents of its send buffer to the root process. The root process receives the messages and stores
11	them in rank order. The outcome is as if each of the n processes in the group (including
12	the root process) had executed a call to
13	<b>x</b> )
14	MPI_Send(sendbuf, sendcount, sendtype, root ,), and the root had executed n calls to
15	
16	$MPI\_Recv(recvbuf+i\cdotrecvcount\cdotextent(recvtype),recvcount,recvtype,i,),\mathrm{where}extent(recvtype)$
17	is the type extent obtained from a call to MPI_Type_get_extent.
18	An alternative description is that the $n$ messages sent by the processes in the group
19	are concatenated in rank order, and the resulting message is received by the root as if by a
20	call to MPI_RECV(recvbuf, recvcount $\cdot$ n, recvtype,).
21	The receive buffer is ignored for all non-root processes.
22	General, derived datatypes are allowed for both sendtype and recvtype. The type signa-
23	ture of sendcount, sendtype on each process must be equal to the type signature of recvcount,
24	recvtype at the root. This implies that the amount of data sent must be equal to the amount
25	of data received, pairwise between each process and the root. Distinct type maps between
26	sender and receiver are still allowed.
27	All arguments to the function are significant on process root, while on other processes,
28	only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
29	root and comm must have identical values on all processes.
30	The specification of counts and types should not cause any location on the root to be
31	written more than once. Such a call is erroneous.
32	Note that the <b>recvcount</b> argument at the root indicates the number of items it receives
33	from <i>each</i> process, not the total number of items it receives.
34	The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as
35	the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and
36	the contribution of the root to the gathered vector is assumed to be already in the correct
37	place in the receive buffer.
38	If comm is an intercommunicator, then the call involves all processes in the intercom-
39	municator, but with one group (group A) defining the root process. All processes in the
40	other group (group B) pass the same value in argument root, which is the rank of the root
41	in group A. The root passes the value MPI_ROOT in root. All other processes in group A
42	pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to
43	the root. The send buffer arguments of the processes in group B must be consistent with
44	the receive buffer argument of the root.
45	
46	
477	

MPI_GATH	IERV(sendbuf, sendcount, sen comm)	dtype, recvbuf, recvcounts, displs, recvtype, root,	$\frac{1}{2}$
IN	sendbuf	starting address of send buffer (choice)	3
			4
IN	sendcount	number of elements in send buffer (non-negative integer)	5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice, significant only at root)	8 9 10
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)	10 11 12 13
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)	14 15 16 17
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	18 19 20
IN	root	rank of receiving process (integer)	21
IN	comm	communicator (handle)	22
			23
int MPI_G	void* recvbuf, const	<pre>if, int sendcount, MPI_Datatype sendtype, int recvcounts[], const int displs[], e, int root, MPI_Comm comm)</pre>	24 25 26
			27
MPI_Gathe		endtype, recvbuf, recvcounts, displs,	28
<b>ΤΥΡ</b> Γ <i>(</i>	<pre>recvtype, root, comm *), DIMENSION(), INTENT</pre>		29 30
		ecvbuf	31
		<pre>punt, recvcounts(*), displs(*), root</pre>	32
	MPI_Datatype), INTENT(IN)	-	33
	MPI_Comm), INTENT(IN) ::		34
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	35
MPI GATHE	RV(SENDBUF, SENDCOUNT, SE	ENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	36
-	RECVTYPE, ROOT, COMM		37
<type< td=""><td>&gt; SENDBUF(*), RECVBUF(*)</td><td></td><td>38</td></type<>	> SENDBUF(*), RECVBUF(*)		38
INTEG	ER SENDCOUNT, SENDTYPE, F	RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,	$\frac{39}{40}$
COMM,	IERROR		40 41
MPI (	GATHERV extends the function	nality of MPI_GATHER by allowing a varying count	42
		<b>nts</b> is now an array. It also allows more flexibility	43
as to where the data is placed on the root, by providing the new argument, displs. 44			44
If comm is an intracommunicator, the outcome is $as if$ each process, including the root $^{45}$			45
process goods a maggage to the root			10

process, sends a message to the root,

 $\mathsf{MPI}\_\mathsf{Send}(\mathsf{sendbuf}, \, \mathsf{sendcount}, \, \mathsf{sendtype}, \, \mathsf{root}, \, \ldots), \;$  and the root executes n receives,

 $\frac{46}{47}$ 

<sup>1</sup> MPI\_Recv(recvbuf+displs[j]· extent(recvtype), recvcounts[j], 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.

<sup>19</sup> If comm is an intercommunicator, then the call involves all processes in the intercom-<sup>20</sup> municator, but with one group (group A) defining the root process. All processes in the <sup>21</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>22</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>23</sup> pass the value MPI\_PROC\_NULL in root. Data is gathered from all processes in group B to <sup>24</sup> the root. The send buffer arguments of the processes in group B must be consistent with <sup>25</sup> the receive buffer argument of the root.

26 27

28

30

 $^{31}$ 

32 33

34

35

36

37

38

39

## 5.5.1 Examples using MPI\_GATHER, MPI\_GATHERV

<sup>29</sup> The examples in this section use intracommunicators.

#### Example 5.2

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

MPI\_Comm comm; int gsize,sendarray[100]; int root, \*rbuf;

...
MPI\_Comm\_size(comm, &gsize);
rbuf = (int \*)malloc(gsize\*100\*sizeof(int));
MPI\_Gather(sendarray, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm);

40 41 42

48

## Example 5.3

Previous example modified — only the root allocates memory for the receive buffer.
 45
 46
 47

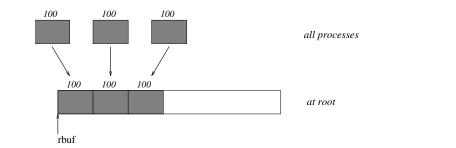


Figure 5.4: The root process gathers 100 ints from each process in the group.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank(comm, &myrank);
if (myrank == root) {
    MPI_Comm_size(comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

# Example 5.4

Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of gsize\*100 ints since type matching is defined pairwise between the root and each process in the gather.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size(comm, &gsize);
MPI_Type_contiguous(100, MPI_INT, &rtype);
MPI_Type_contiguous(100, MPI_INT, &rtype);
mPI_Type_commit(&rtype);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

## Example 5.5

Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use MPI\_GATHERV and the displs argument to achieve this effect. Assume  $stride \geq 100$ . See Figure 5.5.

 $^{24}$ 

```
100
                                  100
                                           100
1
2
                                                                 all processes
3
4
                             100
                                    100
                                           100
5
                                                                  at root
6
7
                                    stride
                           rbuf
8
9
     Figure 5.5: The root process gathers 100 ints from each process in the group, each set is
10
     placed stride ints apart.
11
12
          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);
45
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
46
          displs = (int *)malloc(gsize*sizeof(int));
47
          rcounts = (int *)malloc(gsize*sizeof(int));
48
          for (i=0; i<gsize; ++i) {</pre>
```

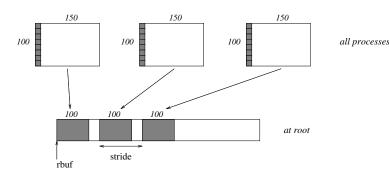


Figure 5.6: The root process gathers column 0 of a  $100 \times 150$  C array, and each set is placed stride ints apart.

```
displs[i] = i*stride;
rcounts[i] = 100;
}
/* Create datatype for 1 column of array
*/
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

# Example 5.7

Process i sends (100-i) ints from the i-th column of a  $100 \times 150$  int array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 5.7.

```
28
MPI_Comm comm;
                                                                                  29
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
                                                                                  30
                                                                                  ^{31}
MPI_Datatype stype;
                                                                                  32
int *displs,i,*rcounts;
                                                                                  33
                                                                                  34
. . .
                                                                                  35
                                                                                  36
MPI_Comm_size(comm, &gsize);
                                                                                  37
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                  38
                                                                                  39
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                  40
                                                                                  41
for (i=0; i<gsize; ++i) {</pre>
                                                                                  42
    displs[i] = i*stride;
    rcounts[i] = 100-i;
                              /* note change from previous example */
                                                                                  43
                                                                                  44
}
/* Create datatype for the column we are sending
                                                                                  45
                                                                                  46
 */
                                                                                  47
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                  48
MPI_Type_commit(&stype);
```

1

 $\mathbf{2}$ 

3 4 5

6

7

9 10

11

12 13

14

15

16

17

18

19

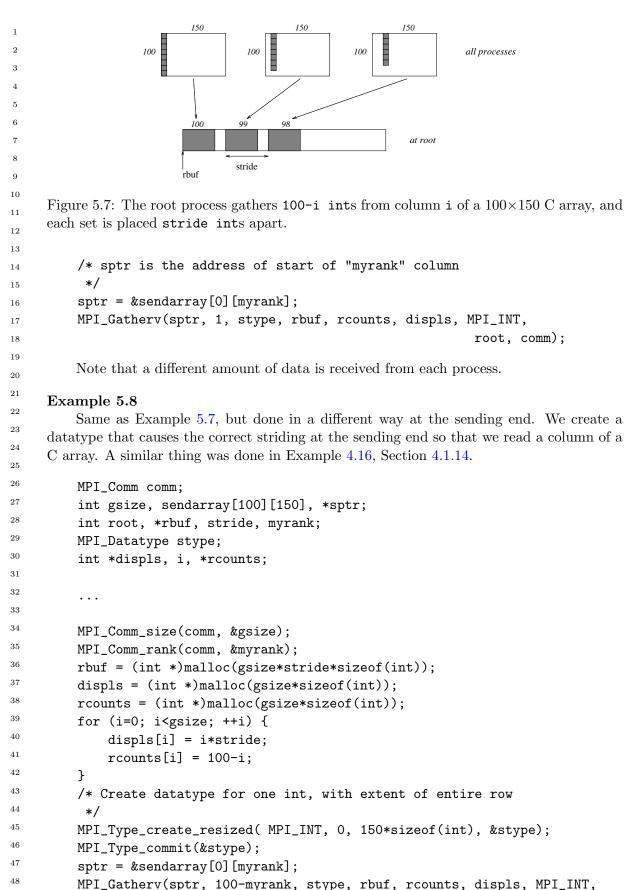
20

21

22 23 24

25

26



2 3

4

5

6 7

root, comm);

```
Example 5.9
```

Same as Example 5.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 5.8.

```
MPI_Comm comm;
                                                                                  8
int gsize,sendarray[100][150],*sptr;
                                                                                  9
int root, *rbuf, *stride, myrank, bufsize;
                                                                                  10
MPI_Datatype stype;
                                                                                  11
int *displs,i,*rcounts,offset;
                                                                                 12
                                                                                  13
. . .
                                                                                 14
                                                                                  15
MPI_Comm_size(comm, &gsize);
                                                                                  16
MPI_Comm_rank(comm, &myrank);
                                                                                  17
                                                                                 18
stride = (int *)malloc(gsize*sizeof(int));
                                                                                 19
. . .
                                                                                 20
/* stride[i] for i = 0 to gsize-1 is set somehow
                                                                                 21
 */
                                                                                 22
                                                                                 23
/* set up displs and rcounts vectors first
                                                                                 ^{24}
 */
                                                                                 25
displs = (int *)malloc(gsize*sizeof(int));
                                                                                  26
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                 27
offset = 0;
                                                                                 28
for (i=0; i<gsize; ++i) {</pre>
                                                                                 29
    displs[i] = offset;
                                                                                 30
    offset += stride[i];
                                                                                 31
    rcounts[i] = 100-i;
                                                                                 32
}
                                                                                 33
/* the required buffer size for rbuf is now easily obtained
                                                                                 34
 */
                                                                                 35
bufsize = displs[gsize-1]+rcounts[gsize-1];
                                                                                 36
rbuf = (int *)malloc(bufsize*sizeof(int));
                                                                                 37
/* Create datatype for the column we are sending
                                                                                 38
 */
                                                                                 39
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                  40
MPI_Type_commit(&stype);
                                                                                 41
sptr = &sendarray[0][myrank];
                                                                                 42
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                                                 43
                                                        root, comm);
                                                                                 44
```

Example 5.10

47 48

 $45 \\ 46$ 

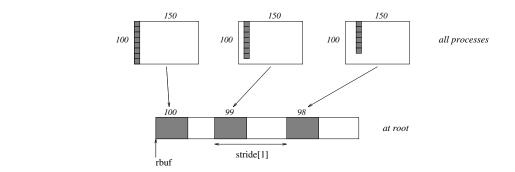


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

18

1

2

7

8

9 10

11

<sup>14</sup> Process i sends num ints from the i-th column of a  $100 \times 150$  int array, in C. The <sup>15</sup> complicating factor is that the various values of num are not known to root, so a separate <sup>16</sup> gather must first be run to find these out. The data is placed contiguously at the receiving <sup>17</sup> end.

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

```
1
    MPI_Type_commit(&stype);
                                                                                            \mathbf{2}
    sptr = &sendarray[0][myrank];
                                                                                            3
    MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                                                                         root, comm);
                                                                                            4
                                                                                            5
                                                                                            6
5.6
      Scatter
                                                                                            7
                                                                                            8
                                                                                            9
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
                                                                                            10
                                                                                            11
            sendbuf
                                        address of send buffer (choice, significant only at root)
  IN
                                                                                            12
            sendcount
  IN
                                        number of elements sent to each process (non-negative
                                                                                            13
                                        integer, significant only at root)
                                                                                            14
                                                                                            15
  IN
            sendtype
                                        data type of send buffer elements (significant only at
                                                                                            16
                                        root) (handle)
                                                                                            17
  OUT
            recvbuf
                                        address of receive buffer (choice)
                                                                                            18
  IN
                                        number of elements in receive buffer (non-negative in-
            recvcount
                                                                                            19
                                        teger)
                                                                                            20
                                                                                            21
  IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                            22
  IN
                                        rank of sending process (integer)
            root
                                                                                            23
  IN
                                        communicator (handle)
            comm
                                                                                            ^{24}
                                                                                            25
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                                                                                            26
               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                                                                                            27
               MPI_Comm comm)
                                                                                            28
                                                                                            29
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                            30
               root, comm, ierror) BIND(C)
                                                                                            31
    TYPE(*), DIMENSION(...), INTENT(IN) ::
                                                  sendbuf
                                                                                            32
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                            33
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                            34
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                            35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                            36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            37
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                            38
               ROOT, COMM, IERROR)
                                                                                            39
    <type> SENDBUF(*), RECVBUF(*)
                                                                                            40
                                                                                            41
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
                                                                                            42
    MPI_SCATTER is the inverse operation to MPI_GATHER.
                                                                                            43
    If comm is an intracommunicator, the outcome is as if the root executed n send oper-
                                                                                            44
ations,
                                                                                            45
                                                                                            46
 MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,...), and each
                                                                                            47
process executed a receive,
                                                                                            48
```

1MPI\_Recv(recvbuf, recvcount, recvtype, i,...).

 $\mathbf{2}$ An alternative description is that the root sends a message with MPI\_Send(sendbuf, 3 sendcount n, sendtype, ...). This message is split into n equal segments, the *i*-th segment is 4 sent to the *i*-th process in the group, and each process receives this message as above.

 $\mathbf{5}$ 6

7

8

9

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 10 between sender and receiver are still allowed.

11All arguments to the function are significant on process root, while on other processes, 12only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments 13root and comm must have identical values on all processes.

14The specification of counts and types should not cause any location on the root to be 15read 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 21the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and 22 root "sends" no data to itself. The scattered vector is still assumed to contain n segments, 23where n is the group size; the *root*-th segment, which root should "send to itself," is not  $^{24}$ moved. 25

If comm is an intercommunicator, then the call involves all processes in the intercom-26municator, but with one group (group A) defining the root process. All processes in the 27other group (group B) pass the same value in argument root, which is the rank of the root 28in group A. The root passes the value MPI\_ROOT in root. All other processes in group A 29pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes in 30 group B. The receive buffer arguments of the processes in group B must be consistent with  $^{31}$ the send buffer argument of the root. 32

- 33
- 34
- 35 36
- 37
- 38
- 39 40
- 41
- 42
- 43
- 4445
- 46
- 47
- 48

16

17

18

IN	_	
	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank
IN	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
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative in-teger)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
	recvtype, root,	unts, displs, sendtype, recvbuf, recvcount, comm, ierror) BIND(C)
	*), DIMENSION(),	
	*), DIMENSION()	<pre>:: recvbuf sendcounts(*), displs(*), recvcount, root</pre>
		NT(IN) :: sendtype, recvtype
	MPI_Comm), INTENT(I	
INTEG	ER, OPTIONAL, INTEN	T(OUT) :: ierror
PI_SCATT	ERV(SENDBUF, SENDCO	UNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
	RECVTYPE, ROOT,	
INTEG	> SENDBUF(*), RECVB ER SENDCOUNTS(*), D IERROR	UF(*) ISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
MPI 9	<b>SCATTERV</b> is the inver-	se operation to MPI_GATHERV.
		e functionality of MPI_SCATTER by allowing a varying
MPI_S	CATTERV extends the	incriticality of Will_SCATTER by anowing a varying

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

 $\mathsf{MPI}\_\mathsf{Send}(\mathsf{sendbuf}+\mathsf{displs}[i]\cdot \mathsf{extent}(\mathsf{sendtype}), \ \mathsf{sendcounts}[i], \ \mathsf{sendtype}, \ \mathsf{i}, \ldots), \ \ \mathrm{and} \ \mathrm{each} \ \mathrm{pro-product}(\mathsf{sendtype}) = \mathsf{sendtype}, \ \mathsf{sen$ cess executed a receive,

<sup>1</sup> 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.

<sup>11</sup> The specification of counts, types, and displacements should not cause any location on <sup>12</sup> the root to be read more than once.

<sup>13</sup> The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE as <sup>14</sup> the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and <sup>15</sup> root "sends" no data to itself. The scattered vector is still assumed to contain n segments, <sup>16</sup> where n is the group size; the *root*-th segment, which root should "send to itself," is not <sup>17</sup> moved.

<sup>18</sup> If comm is an intercommunicator, then the call involves all processes in the intercom-<sup>19</sup> municator, but with one group (group A) defining the root process. All processes in the <sup>20</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>21</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>22</sup> pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes in <sup>23</sup> group B. The receive buffer arguments of the processes in group B must be consistent with <sup>24</sup> the send buffer argument of the root.

25 26

27

 $^{31}$ 

32

## 5.6.1 Examples using MPI\_SCATTER, MPI\_SCATTERV

<sup>28</sup> The examples in this section use intracommunicators.

# <sup>29</sup> 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.

33 MPI\_Comm comm; 34 int gsize,\*sendbuf; 35 int root, rbuf[100]; 36 . . . 37 MPI\_Comm\_size(comm, &gsize); 38 sendbuf = (int \*)malloc(gsize\*100\*sizeof(int)); 39 . . . 40 MPI\_Scatter(sendbuf, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm); 41 42

### <sup>43</sup> Example 5.12

<sup>44</sup> The reverse of Example 5.5. The root process scatters sets of 100 ints to the other <sup>45</sup> processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of <sup>46</sup> MPI\_SCATTERV. Assume *stride*  $\geq$  100. See Figure 5.10.

47 48

 $\mathbf{2}$ 

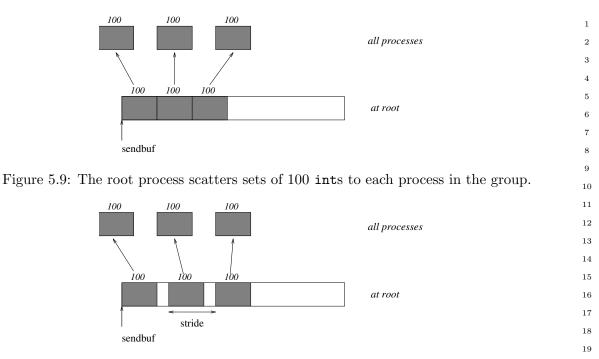


Figure 5.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
```

#### root, comm);

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

 $45 \\ 46$ 

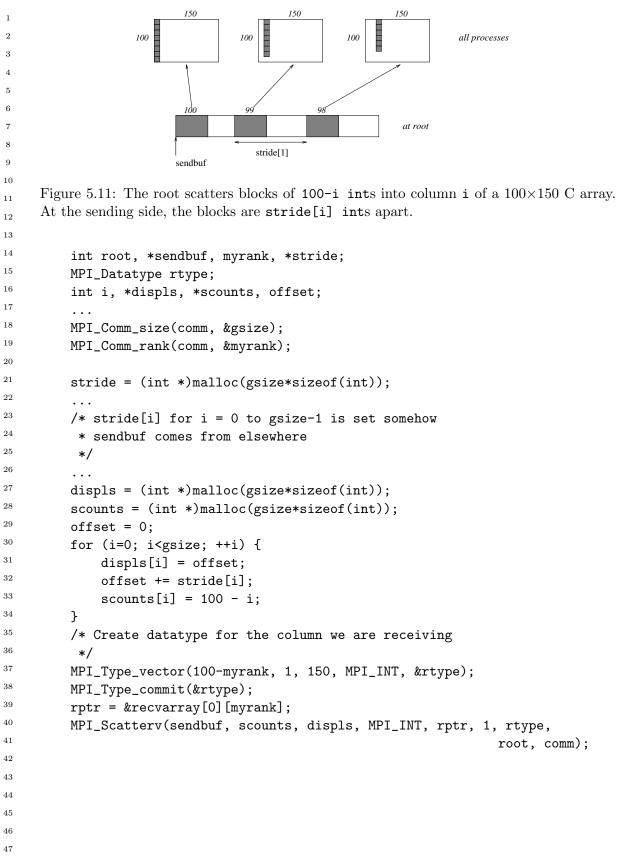
47

48

#### Example 5.13

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  $100 \times 150$  C array. See Figure 5.11.

MPI\_Comm comm; int gsize,recvarray[100][150],\*rptr;



# 5.7 Gather-to-all

5.7 (	Gather-to-all	
MPI_AL	LGATHER(sendbuf, sendo	count, sendtype, recvbuf, recvcount, recvtype, comm)
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
int MPI	[_Allgather(const voi	d* sendbuf, int sendcount,
	MPI_Datatype s	sendtype, void* recvbuf, int recvcount,
	MPI_Datatype r	cecvtype, MPI_Comm comm)
MPI_All	lgather(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype,
-	comm, ierror)	
TYI	PE(*), DIMENSION(),	INTENT(IN) :: sendbuf
TYI	PE(*), DIMENSION()	:: recvbuf
	<pre>FEGER, INTENT(IN) ::</pre>	
	• •	ENT(IN) :: sendtype, recvtype
	PE(MPI_Comm), INTENT(	
INT	FEGER, OPTIONAL, INTE	NT(OUT) :: ierror
MPI_ALI	LGATHER(SENDBUF, SEND COMM, IERROR)	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
•	<pre>ype&gt; SENDBUF(*), RECV</pre>	
INT	FEGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
MP	<b>I_ALLGATHER</b> can be th	nought of as MPI_GATHER, but where all processes receive
		ot. The block of data sent from the j-th process is received
	,	he j-th block of the buffer recvbuf.
		ed with sendcount, sendtype, at a process must be equal to
the type signature associated with recvcount, recvtype at any other process.		
If c	omm is an intracommuni	icator, the outcome of a call to $MPI_ALLGATHER()$ is as
if all pro	presses executed $n$ calls t	0
мрт	Gather(sendbuf.sendc	count,sendtype,recvbuf,recvcount,
··· ±-		recvtype,root,comm)
		rules for correct usage of MPI_ALLGATHER are easily found
from the	e 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.

<sup>3</sup> If comm is an intercommunicator, then each process of one group (group A) contributes <sup>4</sup> sendcount data items; these data are concatenated and the result is stored at each process <sup>5</sup> in the other group (group B). Conversely the concatenation of the contributions of the <sup>6</sup> processes in group B is stored at each process in group A. The send buffer arguments in <sup>7</sup> 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.*)

17 18 19

20 21

8

9

10

11

12

13

14

15

16

MPI\_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

```
IN
                  sendbuf
                                               starting address of send buffer (choice)
22
23
       IN
                  sendcount
                                               number of elements in send buffer (non-negative inte-
^{24}
                                               ger)
25
       IN
                  sendtype
                                               data type of send buffer elements (handle)
26
       OUT
                  recvbuf
                                               address of receive buffer (choice)
27
28
       IN
                                               non-negative integer array (of length group size) con-
                  recvcounts
29
                                               taining the number of elements that are received from
30
                                               each process
^{31}
       IN
                  displs
                                               integer array (of length group size). Entry i specifies
32
                                               the displacement (relative to recvbuf) at which to place
33
                                               the incoming data from process i
34
       IN
                                               data type of receive buffer elements (handle)
                  recvtype
35
36
       IN
                  comm
                                               communicator (handle)
37
38
      int MPI_Allgatherv(const void* sendbuf, int sendcount,
39
                     MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
40
                     const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
41
     MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
42
                     recvtype, comm, ierror) BIND(C)
43
          TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                        sendbuf
44
          TYPE(*), DIMENSION(..) :: recvbuf
45
          INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
46
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
          TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                            \mathbf{2}
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                            3
               RECVTYPE, COMM, IERROR)
                                                                                            4
    <type> SENDBUF(*), RECVBUF(*)
                                                                                            5
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
                                                                                            6
    IERROR
                                                                                            7
                                                                                            8
    MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes re-
                                                                                            9
ceive the result, instead of just the root. The block of data sent from the j-th process is
                                                                                            10
received by every process and placed in the j-th block of the buffer recvbuf. These blocks
                                                                                            11
need not all be the same size.
    The type signature associated with sendcount, sendtype, at process j must be equal to
                                                                                            12
                                                                                            13
the type signature associated with recvcounts[j], recvtype at any other process.
                                                                                            14
    If comm is an intracommunicator, the outcome is as if all processes executed calls to
                                                                                            15
    MPI_Gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
                                                                                            16
                                                           recvtype,root,comm),
                                                                                            17
                                                                                            18
for root = 0, ..., n-1. The rules for correct usage of MPI_ALLGATHERV are easily
                                                                                            19
found from the corresponding rules for MPI_GATHERV.
                                                                                            20
    The "in place" option for intracommunicators is specified by passing the value
                                                                                            21
MPI_IN_PLACE to the argument sendbuf at all processes. In such a case, sendcount and
                                                                                            22
sendtype are ignored, and the input data of each process is assumed to be in the area where
                                                                                            23
that process would receive its own contribution to the receive buffer.
                                                                                            ^{24}
    If comm is an intercommunicator, then each process of one group (group A) contributes
                                                                                            25
sendcount data items; these data are concatenated and the result is stored at each process
                                                                                            26
in the other group (group B). Conversely the concatenation of the contributions of the
                                                                                            27
processes in group B is stored at each process in group A. The send buffer arguments in
                                                                                            28
group A must be consistent with the receive buffer arguments in group B, and vice versa.
                                                                                            29
                                                                                            30
       Example using MPI_ALLGATHER
5.7.1
                                                                                            31
                                                                                            32
The example in this section uses intracommunicators.
                                                                                            33
Example 5.14
                                                                                            34
    The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100
                                                                                            35
ints from every process in the group to every process.
                                                                                            36
                                                                                            37
    MPI_Comm comm;
                                                                                            38
    int gsize, sendarray[100];
                                                                                            39
    int *rbuf;
                                                                                            40
     . . .
                                                                                            41
    MPI_Comm_size(comm, &gsize);
                                                                                            42
    rbuf = (int *)malloc(gsize*100*sizeof(int));
                                                                                            43
    MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
                                                                                            44
                                                                                            45
    After the call, every process has the group-wide concatenation of the sets of data.
                                                                                            46
                                                                                            47
                                                                                            48
```

 $\mathbf{5}$ 

CHAPTER 5. COLLECTIVE COMMUNICATION

# 5.8 All-to-All Scatter/Gather

MPI\_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

6	IN	sendbuf	starting address of send buffer (choice)
7 8	IN	sendcount	number of elements sent to each process (non-negative integer)
9 10	IN	sendtype	data type of send buffer elements (handle)
11	OUT	recvbuf	address of receive buffer (choice)
12 13	IN	recvcount	number of elements received from any process (non-negative integer)
14 15	IN	recvtype	data type of receive buffer elements (handle)
16	IN	comm	communicator (handle)
<ol> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> <li>27</li> <li>28</li> <li>29</li> </ol>	MPI_Alltoa TYPE(* TYPE(* INTEGE TYPE(M TYPE(M	void* recvbuf, int ro MPI_Comm comm)	C(IN) :: sendbuf ecvbuf ount, recvcount :: sendtype, recvtype comm
30 31 32 33 34 35 36 37 38 39 40 41 42 43	<type> INTEGE MPI_A sends distin by process j The ty the type sig that the am every pair o If comr</type>	COMM, IERROR) SENDBUF(*), RECVBUF(*) CR SENDCOUNT, SENDTYPE, F LLTOALL is an extension of act data to each of the receiv and is placed in the i-th blo pe signature associated with gnature associated with recvo ount of data sent must be equ of processes. As usual, however	sendcount, sendtype, at a process must be equal to count, recvtype at any other process. This implies al to the amount of data received, pairwise between er, the type maps may be different. ne outcome is as if each process executed a send to
44 45 46 47 48	from every	other process with a call to,	(sendtype),sendcount,sendtype,i,), and a receive ecvtype),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.*)

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

12	MPI_ALLT(	DALLV(sendbuf, sendcounts, se recvtype, comm)	displs, sendtype, recvbuf, recvcounts, rdispls,
3 4	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank
7 8 9	IN	sdispls	integer array (of length group size). Entry $j$ specifies the displacement (relative to ${\sf sendbuf})$ from which to take the outgoing data destined for process $j$
10 11	IN	sendtype	data type of send buffer elements (handle)
12	OUT	recvbuf	address of receive buffer (choice)
13 14 15	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank
16 17 18 19	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
20	IN	recvtype	data type of receive buffer elements (handle)
21	IN	comm	communicator (handle)
25 26 27 28 29 30 31 32 33 34	TYPE( TYPE() INTEGI rdisp	<pre>int recvcounts[], co MPI_Comm comm) allv(sendbuf, sendcounts rdispls, recvtype, c *), DIMENSION(), INTEN *), DIMENSION() :: re ER, INTENT(IN) :: sendco</pre>	<pre>F(IN) :: sendbuf ecvbuf punts(*), sdispls(*), recvcounts(*),</pre>
35 36	TYPE()	MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)	comm
<ol> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> </ol>	<type INTEGI RECVT MPI_A the send is</type 	RDISPLS, RECVTYPE, C > SENDBUF(*), RECVBUF(*) ER SENDCOUNTS(*), SDISPLS YPE, COMM, IERROR LLTOALLV adds flexibility to	, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, OMM, IERROR) S(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), o MPI_ALLTOALL in that the location of data for ocation of the placement of the data on the receive
46 47 48	If com	<b>n</b> is an intracommunicator, tl	hen the j-th block sent from process i is received by of <b>recvbuf</b> . These blocks need not all have the same

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.)

 $\overline{7}$ 

 $^{24}$ 

12	MPI_ALL	TOALLW(sendbuf, sendcou recvtypes, comm)	nts, sdispls, sendtypes, recvbuf, recvcounts, rdispls,
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank
7 8 9 10	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)
11 12 13	IN	sendtypes	array of datatypes (of length group size). Entry j spec- ifies the type of data to send to process j (array of handles)
14 15	OUT	recvbuf	address of receive buffer (choice)
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 rank
19 20 21 22	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)
23 24 25 26	IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)
27	IN	comm	communicator (handle)
28 29 30 31 32 33		int sdispls[], co const int recvco MPI_Datatype rec	<pre>sendbuf, const int sendcounts[], const onst MPI_Datatype sendtypes[], void* recvbuf, unts[], const int rdispls[], const vtypes[], MPI_Comm comm)</pre>
34 35 36 37 38 39 40 41 42 43 44 45	TYPE TYPE INTE rdis TYPE TYPE INTE	rdispls, recvtyp (*), DIMENSION(), IN (*), DIMENSION() :: GER, INTENT(IN) :: se pls(*) (MPI_Datatype), INTENT (MPI_Comm), INTENT(IN) GER, OPTIONAL, INTENT( OALLW(SENDBUF, SENDCOU	<pre>: recvbuf endcounts(*), sdispls(*), recvcounts(*), f(IN) :: sendtypes(*) f(IN) :: recvtypes(*) 0 :: comm</pre>
46 47 48	INTE	e> SENDBUF(*), RECVBUE	F(*) ISPLS(*), SENDTYPES(*), RECVCOUNTS(*),

#### 5.9. GLOBAL REDUCTION OPERATIONS

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 recvounts and recvtypes 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, and logical and) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

 $^{24}$ 

```
174
                                           CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.9.1
            Reduce
\mathbf{2}
3
4
      MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
5
       IN
                  sendbuf
                                              address of send buffer (choice)
6
       OUT
                  recvbuf
                                              address of receive buffer (choice, significant only at
7
                                              root)
8
9
       IN
                                              number of elements in send buffer (non-negative inte-
                  count
10
                                              ger)
11
       IN
                                              data type of elements of send buffer (handle)
                  datatype
12
       IN
                                              reduce operation (handle)
                  ор
13
14
       IN
                  root
                                              rank of root process (integer)
15
       IN
                  comm
                                              communicator (handle)
16
17
      int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
18
                     MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
19
20
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
21
                     BIND(C)
          TYPE(*), DIMENSION(..), INTENT(IN) ::
22
                                                        sendbuf
23
          TYPE(*), DIMENSION(..) :: recvbuf
^{24}
          INTEGER, INTENT(IN) :: count, root
25
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
          TYPE(MPI_Op), INTENT(IN) :: op
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
30
          <type> SENDBUF(*), RECVBUF(*)
^{31}
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
32
33
          If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the
34
      input buffer of each process in the group, using the operation op, and returns the combined
35
      value in the output buffer of the process with rank root. The input buffer is defined by
36
      the arguments sendbuf, count and datatype; the output buffer is defined by the arguments
37
      recvbuf, count and datatype; both have the same number of elements, with the same type.
38
      The routine is called by all group members using the same arguments for count, datatype, op,
39
      root and comm. Thus, all processes provide input buffers of the same length, with elements
40
      of the same type as the output buffer at the root. Each process can provide one element, or a
41
      sequence of elements, in which case the combine operation is executed element-wise on each
42
      entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains
43
      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)).
44
45
          Section 5.9.2, lists the set of predefined operations provided by MPI. That section also
46
      enumerates the datatypes to which each operation can be applied.
47
          In addition, users may define their own operations that can be overloaded to operate
48
      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 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.

The "in place" option for intracommunicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at the root. In such a case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

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. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root. 43 44 45 46 46 47 88

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

	176	CHAPTER 5. COLLECTIVE COMMUNICATION
1	5.9.2 Predefined Reduction Operat	ions
2 3 4 5 6 7	MPI_ALLREDUCE, MPI_REDUCE_SC MPI_SCAN, MPI_EXSCAN, all nonbl	are supplied for MPI_REDUCE and related functions CATTER_BLOCK, MPI_REDUCE_SCATTER, ocking variants of those (see Section 5.12), and ions are invoked by placing the following in op.
8 9	Name	Meaning
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 23 24 25 26	tion 5.9.4. For the other predefined	<pre>maximum minimum sum product logical and bit-wise and logical or bit-wise or logical exclusive or (xor) bit-wise exclusive or (xor) max value and location min value and location C and MPI_MAXLOC are discussed separately in Secoperations, we enumerate below the allowed combi- s. First, define groups of MPI basic datatypes in the</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39	C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, MPI_UINT64_T MPI_UINT32_T, MPI_UINT64_T
40 41 42 43 44 45 46 47 48	Fortran integer: Floating point:	MPI_INTEGER, and handles returned from MPI_TYPE_CREATE_F90_INTEGER, and if available: MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, MPI_INTEGER16 MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE and handles returned from

	MPI_TYPE_CREATE_F90_REAL,	1
	and if available: MPI_REAL2,	2
	MPI_REAL4, MPI_REAL8, MPI_REAL16	3
Logical:	MPI_LOGICAL,MPI_C_BOOL,	4
-	MPI_CXX_BOOL	5
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	6
	MPI_C_FLOAT_COMPLEX (as synonym),	7
	MPI_C_DOUBLE_COMPLEX,	8
	MPI_C_LONG_DOUBLE_COMPLEX,	9
	MPI_CXX_FLOAT_COMPLEX,	10
	MPI_CXX_DOUBLE_COMPLEX,	11
	MPI_CXX_LONG_DOUBLE_COMPLEX,	11
	and handles returned from	
	MPI_TYPE_CREATE_F90_COMPLEX,	13
	and if available: MPI_DOUBLE_COMPLEX,	14
	MPI_COMPLEX4, MPI_COMPLEX8,	15
	MPI_COMPLEX16, MPI_COMPLEX32	16
Byte:	MPI_BYTE	17
Multi-language types:	MPI_AINT, MPI_OFFSET, MPI_COUNT	18
Multi-language types.		19
Now, the valid datatypes for each	operation are specified below.	20
		21
		22
Op	Allowed Types	23
		24
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	25
	Multi-language types	26
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex,	27
	Multi-language types	28
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	29
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte, Multi-language types	30
These exercisions together with all	listed detetunes are valid in all supported program	
	listed datatypes are valid in all supported program-	31
ming languages, see also Reduce Opera		32
The following examples use intrace	ommunicators.	33
Evennle 5 15		34
Example 5.15		35
	product of two vectors that are distributed across a	36
group of processes and returns the answ	ver at node zero.	37
		38
		39
		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
                              ! result (at node zero)
     REAL c
4
     REAL sum
\mathbf{5}
     INTEGER m, comm, i, ierr
6
\overline{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
       DO 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-
```

tions. MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER (which represent printable characters) 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

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 

 $\mathbf{2}$ 

 $\overline{7}$ 

_			
1		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	and index. For both Fortran and C, types are provided to describe the pair. The potentially		
5		problem in Fortran. The problem is circumvented,	
6		type consist of a pair of the same type as value,	
7	0	. In C, the MPI-provided pair type has distinct	
8	types and the index is an int.		
9		_MAXLOC in a reduce operation, one must provide	
10		pair (value and index). MPI provides nine such	
11		IPI_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	MPI_SHORT_INT	short and int	
28	MPI_LONG_DOUBLE_INT	long double and int	
29	The datatype MPL $2REAL$ is as if det	fined by the following (see Section $4.1$ ).	
30	The datatype with_2REAE is us if de	lined by the following (see Section 4.1).	
31	MPI_TYPE_CONTIGUOUS(2, MPI_REAL, M	PI 2REAL)	
32			
33	Similar statements apply for MPI_2IN	TEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.	
34		f defined by the following sequence of instructions.	
35			
36	type[0] = MPI_FLOAT		
37	type[1] = MPI_INT		
38	disp[0] = 0		
39	disp[1] = sizeof(float)		
40	block[0] = 1		
41	block[1] = 1		
42	MPI_TYPE_CREATE_STRUCT(2, block, d	isp, type, MPI_FLOAT_INT)	
43			
44	Similar statements apply for MPI_LONG_I		
45	The following examples use intracom	nmunicators.	
46	Example 5.17		
47	Example 5.17	blog in C. For each of the 20 leasting accurate	
	Lach process has an array of 30 dou	bles, 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
                                                                                       9
    } in[30], out[30];
                                                                                       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);
    /* At this point, the answer resides on process root
                                                                                       18
                                                                                       19
     */
                                                                                       20
    if (myrank == root) {
                                                                                       21
        /* read ranks out
         */
                                                                                       22
        for (i=0; i<30; ++i) {</pre>
                                                                                       23
             aout[i] = out[i].val;
                                                                                       ^{24}
                                                                                       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];
29
     in.index = 0;
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 advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI\_MAXLOC and MPI\_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

#### 5.9.5 User-Defined Reduction Operations

MPI\_OP\_CREATE(user\_fn, commute, op) user\_fn IN user defined function (function) IN commute true if commutative; false otherwise. OUT operation (handle) op int MPI\_Op\_create(MPI\_User\_function\* user\_fn, int commute, MPI\_Op\* op) MPI\_Op\_create(user\_fn, commute, op, ierror) BIND(C) PROCEDURE(MPI\_User\_function) :: user\_fn LOGICAL, INTENT(IN) :: commute TYPE(MPI\_Op), INTENT(OUT) :: op INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_OP\_CREATE( USER\_FN, COMMUTE, OP, IERROR) EXTERNAL USER\_FN LOGICAL COMMUTE INTEGER OP, IERROR

MPI\_OP\_CREATE binds a user-defined reduction operation to an op handle that can subsequently be used in MPI\_REDUCE, MPI\_ALLREDUCE, MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER, MPI\_SCAN, MPI\_EXSCAN, all nonblocking variants of those (see Section 5.12), and MPI\_REDUCE\_LOCAL. The user-defined operation is assumed to be associative. If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

The argument user\_fn is the user-defined function, which must have the following four arguments: invec, inoutvec, len, and datatype. The ISO C prototype for the function is the following. two defined MDL large function (are identicated on the input terms).

# typedef void MPI\_User\_function(void\* invec, void\* inoutvec, int \*len, MPI\_Datatype \*datatype);

The Fortran declarations of the user-defined function user\_fn appear below. ABSTRACT INTERFACE SUBROUTINE MPI\_User\_function(invec, inoutvec, len, datatype) BIND(C) USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR

1

 $\mathbf{2}$ 

3

4

5

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

43

44

45

46

47

1	
1 2	TYPE(C_PTR), VALUE :: invec, inoutvec
3	INTEGER :: len
4	TYPE(MPI_Datatype) :: datatype
5	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
6	<type> INVEC(LEN), INOUTVEC(LEN)</type>
7	INTEGER LEN, DATATYPE
8	The datatype argument is a handle to the data type that was passed into the call to
9	MPI_REDUCE. The user reduce function should be written such that the following holds:
10	Let $u[0], \ldots, u[len-1]$ be the len elements in the communication buffer described by the
11	arguments invec, len and datatype when the function is invoked; let $v[0], \ldots, v[len-1]$ be len
12	elements in the communication buffer described by the arguments inoutvec, len and datatype
13	when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication
14	buffer described by the arguments inoutvec, len and datatype when the function returns;
15	then $w[i] = u[i] \circ v[i]$ , for i=0,, len-1, where $\circ$ is the reduce operation that the function
16	computes.
17	Informally, we can think of invec and inoutvec as arrays of len elements that user_fn
18	is combining. The result of the reduction over-writes values in inoutvec, hence the name.
19	Each invocation of the function results in the pointwise evaluation of the reduce operator
20	on len elements: i.e., the function returns in $inoutvec[i]$ the value $invec[i] \circ inoutvec[i]$ , for
21	i=0,, count-1, where $\circ$ is the combining operation computed by the function.
22	
23	<i>Rationale.</i> The len argument allows MPI_REDUCE to avoid calling the function for
24 25	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
25 26	with Fortran.
27	
28	By internally comparing the value of the datatype argument to known, global handles,
29	it is possible to overload the use of a single user-defined function for several, different data tamor (End of rationals)
30	data types. (End of rationale.)
31	General datatypes may be passed to the user function. However, use of datatypes that
32	are not contiguous is likely to lead to inefficiencies.
33	No MPI communication function may be called inside the user function. MPI_ABORT
34	may be called inside the function in case of an error.
35	
36	Advice to users. Suppose one defines a library of user-defined reduce functions that
37	are overloaded: the datatype argument is used to select the right execution path at each
38	invocation, according to the types of the operands. The user-defined reduce function
39	cannot "decode" the <b>datatype</b> argument that it is passed, and cannot identify, by itself,
40	the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the
41	
42	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
43	datatypes in global, static variables that are shared by the user code and the library
44	code.
45 46	
40 47	The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using
48	the Fortran calling conventions and will pass a Fortran-type datatype argument; the
10	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);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root
 */
if (rank == root) {
    MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
}
if (rank == groupsize-1) {
    MPI_Send(sendbuf, count, datatype, root, ...);
}
if (rank == root) {
    MPI_Wait(&req, &status);
}
```

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 operations. However, better performance might be achieved if MPI\_REDUCE handles these functions as a special case. (*End of advice to implementors.*)

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
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_OP_FREE(OP, IERROR)
3
          INTEGER OP, IERROR
4
5
         Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
6
7
     Example of User-defined Reduce
8
     It is time for an example of user-defined reduction. The example in this section uses an
9
     intracommunicator.
10
11
     Example 5.20 Compute the product of an array of complex numbers, in C.
12
13
     typedef struct {
14
          double real, imag;
15
     } Complex;
16
17
     /* the user-defined function
18
      */
19
     void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
20
     {
21
          int i;
22
          Complex c;
23
          Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
^{24}
25
          for (i=0; i< *len; ++i) {</pre>
26
              c.real = inout->real*in->real -
27
                           inout->imag*in->imag;
28
              c.imag = inout->real*in->imag +
                           inout->imag*in->real;
29
30
              *inout = c;
^{31}
              in++; inout++;
32
          }
33
     }
34
35
     /* and, to call it...
36
      */
37
      . . .
38
39
          /* each process has an array of 100 Complexes
40
           */
41
          Complex a[100], answer[100];
42
          MPI_Op myOp;
43
          MPI_Datatype ctype;
44
45
          /* explain to MPI how type Complex is defined
46
           */
47
          MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
48
          MPI_Type_commit(&ctype);
```

```
/* create the complex-product user-op
 */
MPI_Op_create( myProd, 1, &myOp );
MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
/* At this point, the answer, which consists of 100 Complexes,
 * resides on process root
 */
```

Example 5.21 How to use the mpi\_f08 interface of the Fortran MPI\_User\_function.

```
subroutine my_user_function( invec, inoutvec, len, type ) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
use mpi_f08
type(c_ptr), value :: invec, inoutvec
integer :: len
type(MPI_Datatype) :: type
real, pointer :: invec_r(:), inoutvec_r(:)
if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
    call c_f_pointer(invec, invec_r, (/ len /) )
    call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
    inoutvec_r = invec_r + inoutvec_r
end if
end subroutine
```

### 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.

MPI\_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)	35
OUT	recvbuf	starting address of receive buffer (choice)	36
IN	count	number of elements in send buffer (non-negative integer)	37 38
IN	datatype	data type of elements of send buffer (handle)	39 40
IN	ор	operation (handle)	41
IN	comm	communicator (handle)	42
			43 44
int MPI_	Allreduce(const void* sen	dbuf, void* recvbuf, int count,	45

MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)

MPI\_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf

 $\mathbf{2}$ 

 $14 \\ 15$ 

 $46 \\ 47$ 

1	
	TYPE(*), DIMENSION() :: recvbuf
2	INTEGER, INTENT(IN) :: count
3	TYPE(MPI_Datatype), INTENT(IN) :: datatype
4	TYPE(MPI_Op), INTENT(IN) :: op
5	TYPE(MPI_Comm), INTENT(IN) :: comm
6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7	
8	MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
9	<type> SENDBUF(*), RECVBUF(*)</type>
10	INTEGER COUNT, DATATYPE, OP, COMM, IERROR
11	If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as
12	MPI_REDUCE except that the result appears in the receive buffer of all the group members.
13	White the post of an one result appears in the receive suffer of an the group members.
14	Advice to implementors. The all-reduce operations can be implemented as a re-
15	duce, followed by a broadcast. However, a direct implementation can lead to better
16	performance. (End of advice to implementors.)
17	r · · · · · · · · · · · · · · · · · · ·
18	The "in place" option for intracommunicators is specified by passing the value
19	MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is
20	taken at each process from the receive buffer, where it will be replaced by the output data.
20	If comm is an intercommunicator, then the result of the reduction of the data provided
21	by processes in group A is stored at each process in group B, and vice versa. Both groups
23	should provide <b>count</b> and <b>datatype</b> arguments that specify the same type signature.
23 24	The following example uses an intracommunicator.
24	
26	Example 5.22
	A routine that computes the product of a vector and an array that are distributed
27	across a group of processes and returns the answer at all nodes (see also Example $5.16$ ).
28	
29	SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
30	REAL a(m), b(m,n) ! local slice of array
31	REAL c(n) ! result
32	REAL sum(n)
32 33	
32 33 34	REAL sum(n) INTEGER n, comm, i, j, ierr
32 33 34 35	REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum
32 33 34 35 36	REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n
32 33 34 35 36 37	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0</pre>
32 33 34 35 36 37 38	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0 DO i = 1, m</pre>
32 33 34 35 36 37 38 39	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n    sum(j) = 0.0    DO i = 1, m       sum(j) = sum(j) + a(i)*b(i,j)</pre>
32 33 34 35 36 37 38	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0 DO i = 1, m</pre>
32 33 34 35 36 37 38 39	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n    sum(j) = 0.0    DO i = 1, m       sum(j) = sum(j) + a(i)*b(i,j)</pre>
32 33 34 35 36 37 38 39 40	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0 DO i = 1, m    sum(j) = sum(j) + a(i)*b(i,j) END DO</pre>
32 33 34 35 36 37 38 39 40 41	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0 DO i = 1, m    sum(j) = sum(j) + a(i)*b(i,j) END DO</pre>
32 33 34 35 36 37 38 39 40 41 42	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0   DO i = 1, m     sum(j) = sum(j) + a(i)*b(i,j)   END DO END DO</pre>
32 33 34 35 36 37 38 39 40 41 42 43	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0   DO i = 1, m     sum(j) = sum(j) + a(i)*b(i,j)   END DO END DO ! global sum</pre>
32 33 34 35 36 37 38 39 40 41 42 43 44	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n   sum(j) = 0.0   DO i = 1, m     sum(j) = sum(j) + a(i)*b(i,j)   END DO END DO ! global sum</pre>
32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum D0 j= 1, n sum(j) = 0.0 D0 i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END D0 END D0 ! global sum CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)</pre>
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<pre>REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum D0 j= 1, n sum(j) = 0.0 D0 i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END D0 END D0 ! global sum CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr) ! return result at all nodes</pre>

5.9.7 Pro	ocess-Local Reduction		1	
implement MPI opera	special reduction patterns th tions.	ortance to library implementors who may want to at are otherwise not easily covered by the standard luction operator to local arguments.	2 3 4 5 6 7	
MPI_REDU	JCE_LOCAL( inbuf, inoutbuf,	count, datatype, op)	8	
IN	inbuf	input buffer (choice)	9 10	
INOUT	inoutbuf	combined input and output buffer (choice)	11	
IN	count	number of elements in inbuf and inoutbuf buffers (non-negative integer)	12 13	
IN	datatype	data type of elements of inbuf and inoutbuf buffers (handle)	14 15 16	
IN	ор	operation (handle)	10 17 18	
int MPI_R	educe_local(const void* MPI_Datatype datatyp	inbuf, void* inoutbuf, int count, e, MPI_Op op)	19 20 21	
<pre>MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: inbuf TYPE(*), DIMENSION() :: inoutbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) <type> INBUF(*), INOUTBUF(*) INTEGER COUNT, DATATYPE, OP, IERROR</type>			29 30 31 32	
The function applies the operation given by <b>op</b> element-wise to the elements of inbuf and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined operations in Section 5.9.5. Both inbuf and inoutbuf (input as well as result) have the same number of elements given by count and the same datatype given by datatype. The MPI_IN_PLACE option is not allowed. Reduction operations can be queried for their commutativity.				
MPI_OP_C	COMMUTATIVE( op, commute	e)	39 40	
IN	ор	operation (handle)	41 42	
OUT	commute	true if op is commutative, false otherwise (logical)	43	
			44	
int MPI_C	p_commutative(MPI_Op op,	int *commute)	45 46	
-	<pre>mmutative(op, commute, i MPI_Op), INTENT(IN) ::</pre>	error) BIND(C) op	40 47 48	

1 2		AL, INTENT(OUT) :: commu ER, OPTIONAL, INTENT(OUT)			
3 4 5 6 7	LOGICA	MUTATIVE(OP, COMMUTE, IE AL COMMUTE ER OP, IERROR	ERROR)		
8 9	5.10 Re	duce-Scatter			
10 11 12 13	in a group o	-	ations where the result is scattered to all processes rs equal-sized blocks to all processes, while another size for each process.		
14 15 16	5.10.1 MF	PI_REDUCE_SCATTER_BL	OCK		
17 18	MPI_REDU	CE_SCATTER_BLOCK( send	buf, recvbuf, recvcount, datatype, op, comm)		
19	IN	sendbuf	starting address of send buffer (choice)		
20 21	OUT	recvbuf	starting address of receive buffer (choice)		
22	IN	recvcount	element count per block (non-negative integer)		
23 24	IN	datatype	data type of elements of send and receive buffers (han- dle)		
25 26	IN	ор	operation (handle)		
27	IN	comm	communicator (handle)		
28 29 30 31	int MPI_Re		; void* sendbuf, void* recvbuf, atatype datatype, MPI_Op op,		
32 33 34 35		e_scatter_block(sendbuf, ierror) BIND(C) <), DIMENSION(), INTENT	recvbuf, recvcount, datatype, op, comm, C(IN) :: sendbuf		
36		<pre>&lt;), DIMENSION() :: re </pre>			
37		ER, INTENT(IN) :: recvcc MPI_Datatype), INTENT(IN)			
$\frac{38}{39}$	TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op				
40		<pre>(PI_Comm), INTENT(IN) ::</pre>			
41	INTEGE	ER, OPTIONAL, INTENT(OUT)	:: lerror		
42 43	MPI_REDUCE		RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,		
44	<tvpe></tvpe>	<pre>IERROR) &gt; SENDBUF(*), RECVBUF(*)</pre>			
45	• -	ER RECVCOUNT, DATATYPE, C	DP, COMM, IERROR		
46 $47$	If comr	n is an intracommunicator, N	$MPI_REDUCE_SCATTER_BLOCK \text{ first performs a}$		
48	global, elem	ent-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.*)

The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE in the sendbuf argument on *all* processes. In this case, the input data is taken from the 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 value for the recvcount argument, and provide input vectors of count =  $n^{recvcount}$  elements stored in the send buffers, where n is the size of the group. The number of elements count must be the same for the two groups. The resulting vector from the other group is scattered in blocks of recvcount elements among the processes in the group.

*Rationale.* The last restriction is needed so that the length of the send buffer of one group can be determined by the local **recvcount** argument of the other group. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

# 5.10.2 MPI\_REDUCE\_SCATTER

MPI\_REDUCE\_SCATTER extends the functionality of MPI\_REDUCE\_SCATTER\_BLOCK such that the scattered blocks can vary in size. Block sizes are determined by the recvcounts array, such that the i-th block contains recvcounts[i] elements.

MPI\_REDUCE\_SCATTER( sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)	36
IIN	Sellabal	starting address of send buner (choice)	37
OUT	recvbuf	starting address of receive buffer (choice)	38
IN	recvcounts	non-negative integer array (of length group size) spec-	39
		ifying the number of elements of the result distributed	40
		to each process.	41
IN	datatype	data type of elements of send and receive buffers (han-	42
IIN	uatatype		43
		dle)	44
IN	ор	operation (handle)	45
IN	comm	communicator (handle)	46
			47

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

```
1
     int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const
2
                     int recvcounts[], MPI_Datatype datatype, MPI_Op op,
3
                     MPI_Comm comm)
4
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
5
                     ierror) BIND(C)
6
          TYPE(*), DIMENSION(...), INTENT(IN) ::
                                                        sendbuf
7
          TYPE(*), DIMENSION(..) :: recvbuf
8
          INTEGER, INTENT(IN) :: recvcounts(*)
9
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
          TYPE(MPI_Op), INTENT(IN) :: op
11
          TYPE(MPI_Comm), INTENT(IN) :: comm
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
15
                     IERROR)
16
          <type> SENDBUF(*), RECVBUF(*)
17
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
18
          If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global,
19
     element-wise reduction on vectors of count = \sum_{i=0}^{n-1} recvcounts[i] elements in the send buffers
20
     defined by sendbuf, count and datatype, using the operation op, where n is the number of
21
     processes in the group of comm. The routine is called by all group members using the
22
     same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as
23
     n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The
^{24}
     blocks are scattered to the processes of the group. The i-th block is sent to process i and
25
     stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.
26
27
           Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equiv-
28
           alent to: an MPI_REDUCE collective operation with count equal to the sum of
29
           recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. How-
30
           ever, a direct implementation may run faster. (End of advice to implementors.)
31
32
          The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in
33
     the sendbuf argument. In this case, the input data is taken from the receive buffer. It is
34
     not required to specify the "in place" option on all processes, since the processes for which
35
     recvcounts[i] == 0 may not have allocated a receive buffer.
36
          If comm is an intercommunicator, then the result of the reduction of the data provided
37
     by processes in one group (group A) is scattered among processes in the other group (group
38
     B), and vice versa. Within each group, all processes provide the same recvcounts argument,
39
     and provide input vectors of count = \sum_{i=0}^{n-1} recvcounts[i] elements stored in the send buffers,
40
     where n is the size of the group. The resulting vector from the other group is scattered in
41
     blocks of recvcounts[i] elements among the processes in the group. The number of elements
42
     count must be the same for the two groups.
43
```

*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.*)

192

44

45

46

5.11. SCAN	193		
5.11 Scan		1	
5.11.1 Inclusive Scan		3	
		4	
		5	
MPI_SCAN(sendbuf, recvbuf, count, datat	type, op, comm)	6	
IN sendbuf	starting address of send buffer (choice)	7 8	
OUT recvbuf	starting address of receive buffer (choice)	9	
IN count	number of elements in input buffer (non-negative in- teger)	10 11	
IN datatype	data type of elements of input buffer (handle)	12 13	
IN op	operation (handle)	13	
IN comm	communicator (handle)	15	
		16	
<pre>int MPI_Scan(const void* sendbuf,</pre>	void* recvbuf, int count,	17	
	e, MPI_Op op, MPI_Comm comm)	18	
MDT Coop (condbuf recubuf court	deteture on comm icrner) BIND(C)	19	
TYPE(*), DIMENSION(), INTENT	<pre>datatype, op, comm, ierror) BIND(C) C(IN) :: sendbuf</pre>	20	
	ecvbuf	21 22	
INTEGER, INTENT(IN) :: count		22	
TYPE(MPI_Datatype), INTENT(IN)	:: datatype	20	
TYPE(MPI_Op), INTENT(IN) :: c		25	
TYPE(MPI_Comm), INTENT(IN) ::	-	26	
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	27	
MPI_SCAN(SENDBUF, RECVBUF, COUNT,	DATATYDE OD COMM IEDDOD)	28	
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	DATATIFE, OF, COMM, TEAROR)	29	
INTEGER COUNT, DATATYPE, OP, C		30	
		31	
	<b>IPI_SCAN</b> is used to perform a prefix reduction on	32	
	peration returns, in the receive buffer of the process	33	
	in the send buffers of processes with ranks $0,\ldots,i$	34	
	coup members using the same arguments for count,	35	

(inclusive). The routine is called by all group members using the same arguments for count, datatype, op and comm, except that for user-defined operations, the same rules apply as for MPI\_REDUCE. The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI\_REDUCE. 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.

This operation is invalid for intercommunicators.

```
194
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.11.2 Exclusive Scan
\mathbf{2}
3
4
     MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)
5
       IN
                 sendbuf
                                              starting address of send buffer (choice)
6
       OUT
\overline{7}
                 recvbuf
                                              starting address of receive buffer (choice)
8
       IN
                 count
                                              number of elements in input buffer (non-negative in-
9
                                              teger)
10
       IN
                                              data type of elements of input buffer (handle)
                 datatype
11
       IN
                                              operation (handle)
                 ор
12
13
       IN
                 comm
                                              intracommunicator (handle)
14
15
     int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
16
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
17
18
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
19
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          TYPE(*), DIMENSION(..) :: recvbuf
20
          INTEGER, INTENT(IN) ::
                                      count
21
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
23
          TYPE(MPI_Op), INTENT(IN) :: op
24
          TYPE(MPI_Comm), INTENT(IN) :: comm
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
27
          <type> SENDBUF(*), RECVBUF(*)
28
          INTEGER COUNT, DATATYPE, OP, COMM, IERROR
29
30
          If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction
^{31}
     on data distributed across the group. The value in recvbuf on the process with rank 0 is
32
     undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process
33
     with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes
34
     with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the
35
     reduction of the values in the send buffers of processes with ranks 0, \ldots, i-1 (inclusive). The
36
     routine is called by all group members using the same arguments for count, datatype, op and
37
     comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE.
38
     The type of operations supported, their semantics, and the constraints on send and receive
39
     buffers, are as for MPI_REDUCE.
40
```

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
 the local contribution. Note that for non-invertable operations such as MPI\_MAX, the
 exclusive scan cannot be computed with the inclusive scan. (*End of rationale.*)

### 5.11.3 Example using MPI\_SCAN

The example in this section uses an intracommunicator.

### Example 5.23

This example uses a user-defined operation to produce a *segmented scan*. A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

values	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$	
logicals	0	0	1	1	1	0	0	1	
result	$v_1$	$v_1 + v_2$	$v_3$	$v_3 + v_4$	$v_3 + v_4 + v_5$	$v_6$	$v_6 + v_7$	$v_8$	

The operator that produces this effect is

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

where

$$w = \begin{cases} u + v & \text{if } i = j \\ v & \text{if } i \neq j \end{cases}.$$

Note that this is a non-commutative operator. C code that implements it is given below.

```
typedef struct {
    double val;
    int log;
} SegScanPair;
/* the user-defined function
*/
void segScan(SegScanPair *in, SegScanPair *inout, int *len,
                                                  MPI_Datatype *dptr)
{
    int i;
    SegScanPair c;
    for (i=0; i< *len; ++i) {</pre>
        if (in->log == inout->log)
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

2 3

1

4

5

6

7

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

 $47 \\ 48$ 

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.

```
4
         int i, base;
5
         SegScanPair
                       a, answer;
6
         MPI_Op
                        myOp;
7
         MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
8
         MPI_Aint
                        disp[2];
9
                        blocklen[2] = \{ 1, 1\};
         int
10
         MPI_Datatype sspair;
11
12
         /* explain to MPI how type SegScanPair is defined
13
           */
14
         MPI_Get_address( &a, disp);
15
         MPI_Get_address( &a.log, disp+1);
16
         base = disp[0];
17
         for (i=0; i<2; ++i) disp[i] -= base;</pre>
18
         MPI_Type_create_struct( 2, blocklen, disp, type, &sspair );
19
         MPI_Type_commit( &sspair );
20
         /* create the segmented-scan user-op
21
          */
22
         MPI_Op_create(segScan, 0, &myOp);
23
         . . .
24
         MPI_Scan( &a, &answer, 1, sspair, myOp, comm );
25
```

# 5.12 Nonblocking Collective Operations

29As described in Section 3.7, performance of many applications can be improved by over-30 lapping communication and computation, and many systems enable this. Nonblocking  $^{31}$ collective operations combine the potential benefits of nonblocking point-to-point opera-32 tions, to exploit overlap and to avoid synchronization, with the optimized implementation 33 and message scheduling provided by collective operations [30, 34]. One way of doing this 34would be to perform a blocking collective operation in a separate thread. An alternative 35 mechanism that often leads to better performance (e.g., avoids context switching, scheduler 36 overheads, and thread management) is to use nonblocking collective communication [32].

37 The nonblocking collective communication model is similar to the model used for non-38blocking point-to-point communication. A nonblocking call initiates a collective operation, 39 which must be completed in a separate completion call. Once initiated, the operation 40may 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 43enabling communication-computation overlap, nonblocking collective operations can per-44form collective operations on overlapping communicators, which would lead to deadlocks 45with blocking operations. Their semantic advantages can also be useful in combination with 46 point-to-point communication.

<sup>47</sup> As in the nonblocking point-to-point case, all calls are local and return immediately, <sup>48</sup> irrespective of the status of other processes. The call initiates the operation, which indicates

1

 $\mathbf{2}$ 

3

26 27

that the system may start to copy data out of the send buffer and into the receive buffer. Once initiated, all associated send buffers and buffers associated with input arguments (such as arrays of counts, displacements, or datatypes in the vector versions of the collectives) should not be modified, and all associated receive buffers should not be accessed, until the collective operation completes. The call returns a request handle, which must be passed to a completion call.

All completion calls (e.g., MPI\_WAIT) described in Section 3.7.3 are supported for nonblocking collective operations. Similarly to the blocking case, nonblocking collective operations are considered to be complete when the local part of the operation is finished, i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be safely accessed and modified. Completion does not indicate that other processes have completed or even started the operation (unless otherwise implied by the description of the operation). Completion of a particular nonblocking collective operation also does not indicate completion of any other posted nonblocking collective (or send-receive) operations, whether they are posted before or after the completed operation.

Advice to users. Users should be aware that implementations are allowed, but 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 completes, the MPI\_ERROR field in the associated status object is set appropriately, see Section 3.2.5 on page 30. The values of the MPI\_SOURCE and MPI\_TAG fields are undefined. It is valid to mix different request types (i.e., any combination of 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 a nonblocking collective operation. Nonblocking collective requests are not persistent.

*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.*)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and generate an MPI exception. Quality implementations of MPI should ensure that this happens only in pathological cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with the ordering rules for blocking collective operations in threaded environments.

*Rationale.* Matching blocking and nonblocking collective operations is not allowed because the implementation might use different communication algorithms for the two

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

42

43

44

45

 $46 \\ 47$ 

	198	CH	HAPTER 5.	COLLECTIVE COMMUNICATION
1 2 3	tio		operations 1	ptimized for minimal time to comple- nay balance time to completion with
4 5 6		the use of tags for collective operation of rationale.)	ations can pr	event certain hardware optimizations.
7 8 9 10 11	$\operatorname{col}$ im	lective operations, then a none	olocking colle	re matching blocking and nonblocking ective operation can be initiated and o emulate blocking behavior. ( <i>End of</i>
12 13 14 15 16 17 18 19	as its blocking as both as	ocking counterpart for intracom www.ewise, upon completion, nonblood their blocking counterparts, ar n orders apply. e use of the "in place" option is collective operations. When using send and receive buffers. Such b	municators a cking collection ad the same allowed exact ing the "in p	ollective operation has the same effect and intercommunicators after comple- ve reduction operations have the same restrictions and recommendations on tly as described for the corresponding lace" option, message buffers function I not be modified or accessed until the
20 21 22 23	Pro	n completes. gression rules for nonblocking o king point-to-point operations, r	-	erations are similar to progression of on $3.7.4$ .
24 25 26 27	loc	÷	g nonblockin	re operations can be implemented with g point-to-point communication and a <i>ntors.</i> )
28 29 30	5.12.1	Nonblocking Barrier Synchroniz	zation	
31 32	MPI_IBA	ARRIER(comm , request)		
33	IN	comm	communicat	for (handle)
34 35	OUT	request	communicat	tion request (handle)
36 37	int MPI	_Ibarrier(MPI_Comm comm, MH	PI_Request	*request)
38 39 40 41 42	TYP TYP INT	rrier(comm, request, ierron E(MPI_Comm), INTENT(IN) :: E(MPI_Request), INTENT(OUT) EGER, OPTIONAL, INTENT(OUT)	comm ) :: reque ) :: ierro	
42 43 44		RRIER(COMM, REQUEST, IERROF EGER COMM, REQUEST, IERROR	3)	
45 46 47 48	a proces	s notifies that it has reached th	e barrier. T	BARRIER. By calling MPI_IBARRIER, the call returns immediately, indepen- ARRIER. The usual barrier semantics

are enforced at the corresponding completion operation (test or wait), which in the intracommunicator 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.

Advice to users. A nonblocking barrier can be used to hide latency. Moving independent computations between the MPI\_IBARRIER and the subsequent completion call can overlap the barrier latency and therefore shorten possible waiting times. The semantic properties are also useful when mixing collective operations and point-to-point messages. (*End of advice to users.*)

### 5.12.2 Nonblocking Broadcast

MPI\_IBCAST(buffer, count, datatype, root, comm, request)

INOUT	buffer	starting address of buffer (choice)	17	
IN	count	number of entries in buffer (non-negative integer)	18	
IN	datatype	data type of buffer (handle)	19	
IN	root	rank of broadcast root (integer)	20 21	
IN	comm	communicator (handle)	21	
			23	
OUT	request	communication request (handle)	24	
int MDT	These t/and dy huffers int	count MDT Deteture deteture int most	25	
int MPI	MPI_Comm comm, MPI_F	count, MPI_Datatype datatype, int root,	26	
		request *request)	27	
<pre>MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C)</pre>				
	(*), DIMENSION(), ASYNC		29	
	GER, INTENT(IN) :: count		30 31	
	(MPI_Datatype), INTENT(IN (MPI_Comm), INTENT(IN) ::		32	
	(MPI_COMM), INTENT(IN) (MPI_Request), INTENT(OUT		33	
	GER, OPTIONAL, INTENT(OUT	-	34	
			35	
		E, ROOT, COMM, REQUEST, IERROR)	36	
<i>v</i> 1	∍> BUFFER(*) GER COUNT, DATATYPE, ROOT		37	
	JER COONI, DATATIPE, ROOT	, COMM, REQUEST, TERROR	38	
This	call starts a nonblocking varia	ant of $MPI_BCAST$ (see Section 5.4).	39	
			40	
Example u	sing MPI_IBCAST		41 42	
			-14	

The example in this section uses an intracommunicator.

### Example 5.24

Start a broadcast of 100 ints from process 0 to every process in the group, perform some computation on independent data, and then complete the outstanding broadcast operation.

199

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10 11 12

13 14 15

16

47 48

43 44

45

1 2 3 4 5 6 7 8 9 10	int MPI  MPI com	_Comm comm; array1[100], arr root=0; _Request req; _Ibcast(array1, 1 pute(array2, 100) _Wait(&req, MPI_S Nonblocking Gather	00, MPI_INT, root, comm, &req); ; TATUS_IGNORE);
11 12	5.12.5	Nonbiocking Gatilei	
12 13 14 15	MPI_IGA	THER(sendbuf, send request)	count, sendtype, recvbuf, recvcount, recvtype, root, comm,
16	IN	sendbuf	starting address of send buffer (choice)
17 18	IN	sendcount	number of elements in send buffer (non-negative integer)
19 20	IN	sendtype	data type of send buffer elements (handle)
20 21 22	OUT	recvbuf	address of receive buffer (choice, significant only at root)
23 24	IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
25 26 27	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
28	IN	root	rank of receiving process (integer)
29	IN	comm	communicator (handle)
$30 \\ 31$	OUT	request	communication request (handle)
32 33 34 35	int MPI	void* recvb	id* sendbuf, int sendcount, MPI_Datatype sendtype, ouf, int recvcount, MPI_Datatype recvtype, int root, omm, MPI_Request *request)
36	MPI_Iga		dcount, sendtype, recvbuf, recvcount, recvtype,
37 38	TVD		, request, ierror) BIND(C) .), INTENT(IN), ASYNCHRONOUS :: sendbuf
39			.), ASYNCHRONOUS :: recvbuf
40			:: sendcount, recvcount, root
41		• -	<pre>INTENT(IN) :: sendtype, recvtype</pre>
42 43		E(MPI_Comm), INTE	
44		-	NTENT(OUT) :: request NTENT(OUT) :: ierror
45			
46	MP1_1GA		DCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, , REQUEST, IERROR)
47 48	<ty< th=""><th>pe&gt; SENDBUF(*), R</th><th></th></ty<>	pe> SENDBUF(*), R	

	ITEGER SENDCOUNT, SENDT ERROR	YPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	1 2 3	
Tl	nis call starts a nonblocking	g variant of $MPI_GATHER$ (see Section 5.5).	4	
			5	
MPI_IC	GATHERV(sendbuf, sendcou comm, request)	int, sendtype, recvbuf, recvcounts, displs, recvtype, root,	6 7	
IN	sendbuf	starting address of send buffer (choice)	8 9	
IN	sendcount	number of elements in send buffer (non-negative integer)	10 11	
IN	sendtype	data type of send buffer elements (handle)	12	
OUT	recvbuf	address of receive buffer (choice, significant only at root)	13 14 15	
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)	16 17 18	
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)	19 20 21 22 23	
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	23 24 25	
IN	root	rank of receiving process (integer)	26	
IN	comm	communicator (handle)	27	
OUT	request	communication request (handle)	28 29	
int MF	<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>			
MPI_Ig	gatherv(sendbuf, sendco	unt, sendtype, recvbuf, recvcounts, displs,	35	
	• -	comm, request, ierror) BIND(C)	36 37	
		INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf	38	
	<pre>ITEGER, INTENT(IN) ::</pre>		39	
		NCHRONOUS :: recvcounts(*), displs(*)	40 41	
		NT(IN) :: sendtype, recvtype	42	
	YPE(MPI_Comm), INTENT(I YPE(MPI_Request), INTEN		43	
	TEGER, OPTIONAL, INTEN	-	44	
		UNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	45 46	
		COMM, REQUEST, IERROR)	40 47	
<t< td=""><td><pre>xype&gt; SENDBUF(*), RECVB</pre></td><td></td><td>48</td></t<>	<pre>xype&gt; SENDBUF(*), RECVB</pre>		48	

```
202
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
\mathbf{2}
          COMM, REQUEST, IERROR
3
         This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).
4
5
     5.12.4 Nonblocking Scatter
6
7
8
9
     MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
10
                     request)
11
       IN
                  sendbuf
                                             address of send buffer (choice, significant only at root)
12
       IN
                  sendcount
                                             number of elements sent to each process (non-negative
13
                                             integer, significant only at root)
14
15
       IN
                  sendtype
                                             data type of send buffer elements (significant only at
16
                                             root) (handle)
17
       OUT
                  recvbuf
                                             address of receive buffer (choice)
18
       IN
                                             number of elements in receive buffer (non-negative in-
                  recvcount
19
                                             teger)
20
       IN
                                             data type of receive buffer elements (handle)
21
                  recvtype
22
       IN
                  root
                                             rank of sending process (integer)
23
       IN
                                             communicator (handle)
                  comm
^{24}
       OUT
25
                 request
                                             communication request (handle)
26
27
     int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
28
                     void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
29
                    MPI_Comm comm, MPI_Request *request)
30
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
^{31}
                    root, comm, request, ierror) BIND(C)
32
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
33
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
34
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
35
          TYPE(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
40
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
41
                    ROOT, COMM, REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
44
          IERROR
45
         This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).
46
47
48
```

MPI_I	SCATTERV(sendbuf, sendcounts, comm, request)	displs, sendtype, recvbuf, recvcount, recvtype, root,	1 $2$		
IN	sendbuf	address of send buffer (choice, significant only at root)	3		
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	4 5 6		
IN	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	7 8 9		
IN	sendtype	data type of send buffer elements (handle)	10 11		
OUT	recvbuf	address of receive buffer (choice)	12		
IN	recvcount	number of elements in receive buffer (non-negative in-teger)	13 14		
IN	recvtype	data type of receive buffer elements (handle)	15 16		
IN	root	rank of sending process (integer)	10		
IN	comm	communicator (handle)	18		
OUT	request	communication request (handle)	19 20		
int M		dbuf, const int sendcounts[], const	22		
<pre>int displs[], MPI_Datatype sendtype, void* recvbuf,</pre>					
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)					
NDT T			25 26		
MP1_1		s, displs, sendtype, recvbuf, recvcount, n, request, ierror) BIND(C)	20		
т	• =	IT(IN), ASYNCHRONOUS :: sendbuf	28		
	YPE(*), DIMENSION(), ASYNC		29		
		<pre>DNOUS :: sendcounts(*), displs(*)</pre>	30		
		count, root	31		
Т	YPE(MPI_Datatype), INTENT(IN	I) :: sendtype, recvtype	32		
	YPE(MPI_Comm), INTENT(IN) ::		33		
	YPE(MPI_Request), INTENT(OUT	-	34		
I	NTEGER, OPTIONAL, INTENT(OUT	:) :: ierror	35 36		
MPI_I	SCATTERV(SENDBUF, SENDCOUNTS	, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	37		
	RECVTYPE, ROOT, COM	M, REQUEST, IERROR)	38		
<	type> SENDBUF(*), RECVBUF(*)		39		
		S(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	40		
C	OMM, REQUEST, IERROR		41		
Γ	his call starts a nonblocking varia	ant of MPI_SCATTERV (see Section $5.6$ ).	42		
	<u> </u>	× /	43		
			44		

	204		CHAPTER 5. COLLECTIVE COMMUNICATION	
1 2 3	5.12.5	Nonblocking Gather-to-al	I	
4 5	MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)			
6 7	IN	sendbuf	starting address of send buffer (choice)	
8 9	IN	sendcount	number of elements in send buffer (non-negative integer)	
10	IN	sendtype	data type of send buffer elements (handle)	
11 12	OUT	recvbuf	address of receive buffer (choice)	
13 14	IN	recvcount	number of elements received from any process (non-negative integer)	
15	IN	recvtype	data type of receive buffer elements (handle)	
16 17	IN	comm	communicator (handle)	
18	OUT	request	communication request (handle)	
21 22 23 24 25 26 27 28 29 30 31 32 33 34	MPI_Ia TYI TYI IN TYI TYI TYI IN	MPI_Datatype sen MPI_Datatype red comm, request, red PE(*), DIMENSION(), I PE(*), DIMENSION(), A FEGER, INTENT(IN) :: s PE(MPI_Datatype), INTEN PE(MPI_Comm), INTENT(IN PE(MPI_Request), INTENT FEGER, OPTIONAL, INTENT	<pre>NTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf endcount, recvcount T(IN) :: sendtype, recvtype ) :: comm (OUT) :: request (OUT) :: ierror OUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,</pre>	
35 36 37 38 39 40 41 42 43 44 45 46 47 48	IN'	-	F(*) PE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR variant of MPI_ALLGATHER (see Section 5.7).	

<pre>N sendbuf starting address of send buffer (choice) N sendcount unmber of elements in send buffer (non-negative inte- ger) N sendtype data type of send buffer clements (handle) OUT recvbuf address of receive buffer (choice) N recvcounts non-negative integer array (of length group size) con- taining the number of elements that are received from each process 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 IN comm communication request (handle) OUT request communication request (handle) IN comm communication request (handle) IN recvtype data type of receive buffer clements (handle) IN comm communication request (handle) OUT request communication request (handle) IN Comm communication request (handle) IN PI_Datatype sendtype, void* recvbuf, recvcounts [], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request) MPI_allgatherv(sendbuf, .), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount INTEGER, INTENT(IN) :: sendcount INTEGER, INTENT(IN) :: sendcount (integer, INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: request INTEGER, SENDGUUNT, SENDTYPE, RECVCOUNTS, NECVTYPE, COMM, REQUEST, IERROR Ctype (Sendbuf, NECVTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7).</pre>	MPI_IAL	LGATHERV(sendbuf, sei request)	ndcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm,	1 $2$
<pre>N sendcount number of elements in send buffer (non-negative inte- ger) N sendtype data type of send buffer elements (handle) OUT recvbuf address of receive buffer (choice) N recvcounts non-negative integer array (of length group size) con- taining the number of elements that are received from cach process 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 IN recvtype data type of receive buffer elements (handle) IN comm communicator (handle) OUT request communication request (handle) int MPI_lallgatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request) MPI_sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendcount INTEGER, INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Request), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONL, INTENT(IN) :: request INTEGER, OPTIONL, INTENT(IN) :: request INTEGER, OPTIONL, INTENT(IN) :: i sendtype, recvtype TYPE(MPI_Request), INTENT(IN) :: i sendtype, RECVEOUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) </pre>	IN	sendbuf	starting address of send buffer (choice)	
IN       Sendype       data type of send tourie recluses (nambe)         OUT       recvbuf       address of receive buffer (choice)         IN       recvcounts       non-negative integer array (of length group size) con- taining the number of elements that are received from each process         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         IN       recvtype       data type of receive buffer elements (handle)         IN       comm       communicator (handle)         OUT       request       communicator request (handle)         IN       const int displs [], MPI_Datatype recvtype, MPI_Comm comm, MPI_Datatype sendtype, void* recvbuf, recvcounts, displs, recvtype, comm, request, ierror) BIND(C)         TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf       29         TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf       29         INTEGER, INTENT(IN), SYNCHRONOUS :: recvount       30         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvount       30         TYPE(MPI_Request), INTENT(IN) :: sendtype, recvtype       31         TYPE(MPI_Request), INTENT(IN) :: sendtype, recvtype       33         TYPE(MPI_Request), INTENT(IN) :: request       33         INTEGER, OPTIONAL, INTENT(OUT) :: request       33         INTEGER, OPTIONAL, INTENT(OUT)	IN	sendcount		5
OUT       recvbuf       address of receive buffer (choice)         IN       recvcounts       non-negative integer array (of length group size) containing the number of elements that are received from each process         IN       displs       integer array (of length group size). Entry i specifies         IN       displs       integer array (of length group size). Entry i specifies         IN       displs       integer array (of length group size). Entry i specifies         IN       recvtype       data type of receive buffer elements (handle)         IN       comm       communicator (handle)         OUT       request       communication request (handle)         OUT       request       communication request (handle)         int MPI_lallgatherv(const void* sendbuf, int sendcount,       22         MPI_Request* request)       23         MPI_Request* request)       24         MPI_NENSION(), ASYNCHRONOUS :: recvbuf       25         TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf       25         TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf       26         TYPE(MPI_datype), INTENT(IN) :: sendtype, recvtype       27         TYPE(MPI_Request), INTENT(OUT) :: request       34         INTEGER, OPTIONAL, INTENT(OUT) :: sendtype, recvtype       35         TYPE(MPI_Request), INTENT(OUT) :	IN	sendtype	data type of send buffer elements (handle)	7
<pre>N recvcounts non-negative integer array (of length group size) con- taining the number of elements that are received from each process i integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i iN recvtype data type of receive buffer elements (handle) iN comm communicator (handle) OUT request communication request (handle) int MPI_Iallgatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request) MPI_lallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount INTEGER, INTENT(IN), ASYNCHRONOUS :: recvtype TYPE(MPI_Comm, INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(IN) :: ierror MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVEOUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7). #### Common c</type></pre>	OUT	recvbuf	address of receive buffer (choice)	
<pre>taining the number of elements that are received from each process 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 IN recvtype data type of receive buffer elements (handle) IN comm communicator (handle) OUT request communication request (handle) int MPI_Iallgatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request) MPI_sign(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvtouf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendcount INTEGER, INTENT(IN), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_CMPI_COMDUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <tp>MPI_IALLGATHERV(SENDEUF, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</tp></pre>	IN	recvcounts	non-negative integer array (of length group size) con-	
<pre>IN Usp15 Integrating (in length group stee). Entry Tayletines the displacement (relative to recvbuf) at which to place the incoming data from process i IN recvtype data type of receive buffer elements (handle) IN comm communicator (handle) OUT request communication request (handle) int MPI_lallgatherv(const void* sendbuf, int sendcount,</pre>			taining the number of elements that are received from	11
<pre>IN recvtype data type of receive buffer elements (handle) IN comm communicator (handle) OUT request communication request (handle) OUT request communication request (handle) int MPI_Batatype sendtype, void* recvbuf, const int recvcounts[],     const int displs[], MPI_Datatype recvtype, MPI_Comm comm,     MPI_Request* request) MPI_lallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,     recvtype, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: ierror MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,     RECVTYPE, COMM, REQUEST, IERROR) 44 This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7).</pre>	IN	displs	the displacement (relative to $recvbuf)$ at which to place	14 15
<pre>IN comm communicator (handle) 18 OUT request communication request (handle) OUT request communication request (handle) int MPI_lallgatherv(const void* sendbuf, int sendcount,</pre>	IN	recvtype	data type of receive buffer elements (handle)	
<pre>001 Tequest communication request (mainter) int MPI_Iallgatherv(const void* sendbuf, int sendcount,</pre>	IN	comm	communicator (handle)	
<pre>int MPI_Iallgatherv(const void* sendbuf, int sendcount,</pre>	OUT	request	communication request (handle)	19
45	<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],</pre>			
				46

1 5.12.6 Nonblocking All-to-All Scatter/Gather  $\mathbf{2}$ 3 4 MPI\_IALLTOALL(sendbuf, sendcount, sendtype, recybuf, recycount, recytype, comm, request) 56 IN sendbuf starting address of send buffer (choice) 7 IN sendcount 8 number of elements sent to each process (non-negative 9 integer) 10 IN sendtype data type of send buffer elements (handle) 11 OUT recvbuf address of receive buffer (choice) 12IN number of elements received from any process (non-13 recvcount 14negative integer) 15IN recvtype data type of receive buffer elements (handle) 16IN communicator (handle) comm 17OUT 18 request communication request (handle) 1920int MPI\_Ialltoall(const void\* sendbuf, int sendcount, 21MPI\_Datatype sendtype, void\* recvbuf, int recvcount, 22 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 23MPI\_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,  $^{24}$ comm, request, ierror) BIND(C) 25TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 26TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 27INTEGER, INTENT(IN) :: sendcount, recvcount 28TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 29 TYPE(MPI\_Comm), INTENT(IN) :: comm 30 TYPE(MPI\_Request), INTENT(OUT) :: request  $^{31}$ INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 MPI\_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 34COMM, REQUEST, IERROR) 35 <type> SENDBUF(\*), RECVBUF(\*) 36 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 37 This call starts a nonblocking variant of MPI\_ALLTOALL (see Section 5.8). 38 39 40 41 4243 44 4546 4748

MPI	IALLTOALLV(sendbuf, sendcounts, s recvtype, comm, request)	sdispls, sendtype, recvbuf, recvcounts, rdispls, )	$\frac{1}{2}$		
IN	sendbuf	starting address of send buffer (choice)	3		
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	4 5 6		
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	7 8 9		
IN	sendtype	data type of send buffer elements (handle)	10 11		
O	JT recvbuf	address of receive buffer (choice)	12		
IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank	13 14 15		
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	16 17 18 19		
IN	recvtype	data type of receive buffer elements (handle)	20		
IN	comm	communicator (handle)	21		
Ol	JT request	communication request (handle)	22 23		
	<pre>int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const</pre>				
	<pre>rdispls, recvtype, c TYPE(*), DIMENSION(), INTEN</pre>	omm, request, ierror) BIND(C) T(IN), ASYNCHRONOUS :: sendbuf	30 31		
	TYPE(*), DIMENSION(), ASYNC	-	32		
		NOUS :: sendcounts(*), sdispls(*),	33		
	<pre>recvcounts(*), rdispls(*) TYDE(MDI Deteture) INTENT(IN)</pre>	) condtime recentime	$\frac{34}{35}$		
	<pre>TYPE(MPI_Datatype), INTENT(IN TYPE(MPI_Comm), INTENT(IN) ::</pre>	VI VI	36		
	TYPE(MPI_Request), INTENT(OUT)		37		
	INTEGER, OPTIONAL, INTENT(OUT	) :: ierror	38		
MPI_	RDISPLS, RECVTYPE, C	S, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, COMM, REQUEST, IERROR)	39 40 41		
	<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</type>				
	This call starts a nonblocking varia	nt of MPI_ALLTOALLV (see Section $5.8$ ).	44 $45$ $46$ $47$		
			48		

# MPLIALITOALIV(sendbuf sendcounts sdispls sendtype recybuf recycounts rdispls 1

12	MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)				
3	IN	sendbuf	starting address of send buffer (choice)		
4 5 6 7	IN	sendcounts	integer array (of length group size) specifying the num- ber of elements to send to each rank (array 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	IN	sendtypes	array of datatypes (of length group size). Entry j spec- ifies the type of data to send to process j (array of handles)		
15 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 rank (array of non-negative integers)		
20 21 22 23	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)		
24 25 26	IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)		
27 28	IN	comm	communicator (handle)		
29	OUT	request	communication request (handle)		
30 31 32 33 34 35	int MPI_	<pre>int sdispls[], cons   const int recvcount</pre>	endbuf, const int sendcounts[], const t MPI_Datatype sendtypes[], void* recvbuf, s[], const int rdispls[], const pes[], MPI_Comm comm, MPI_Request *request)		
36 37 38 39 40 41 42 43 44 45 46	<pre>MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,</pre>				
47 48	MPI_IALL	-	IS, SDISPLS, SENDTYPES, RECVBUF, , RECVTYPES, COMM, REQUEST, IERROR)		

<type> SENDBUF(\*), RECVBUF(\*) 1 2 INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPES(\*), RECVCOUNTS(\*), 3 RDISPLS(\*), RECVTYPES(\*), COMM, REQUEST, IERROR 4 This call starts a nonblocking variant of MPI\_ALLTOALLW (see Section 5.8). 5 6 5.12.7 Nonblocking Reduce 7 9 MPI\_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request) 10 11 IN sendbuf address of send buffer (choice) 12OUT recvbuf address of receive buffer (choice, significant only at 13 root) 14IN number of elements in send buffer (non-negative intecount 1516ger) 17IN data type of elements of send buffer (handle) datatype 18 IN reduce operation (handle) op 19 IN rank of root process (integer) root 2021IN comm communicator (handle) 22 OUT request communication request (handle) 2324int MPI\_Ireduce(const void\* sendbuf, void\* recvbuf, int count, 25MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm, 26MPI\_Request \*request) 2728 MPI\_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, 29ierror) BIND(C) 30 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 31TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 32 INTEGER, INTENT(IN) :: count, root 33 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 34 TYPE(MPI\_Op), INTENT(IN) :: op 35 TYPE(MPI\_Comm), INTENT(IN) :: comm 36 TYPE(MPI\_Request), INTENT(OUT) :: request 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI\_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, 39 IERROR) 40 <type> SENDBUF(\*), RECVBUF(\*) 41 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR 4243 This call starts a nonblocking variant of MPI\_REDUCE (see Section 5.9.1). 44The implementation is explicitly allowed to use different 45Advice to implementors. 46

algorithms for blocking and nonblocking reduction operations that might change the order of evaluation of the operations. However, as for MPI\_REDUCE, it is strongly recommended that MPI\_IREDUCE be implemented so that the same result be obtained 48

	210	CH	HAPTER 5.	COLLECTIVE COMMUNICATION		
1 2 3 4	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 processes. ( <i>End of advice to implementors.</i> )					
5 6 7 8 9	upon ered	Advice to users. For operations which are not truly associative, the result delivered upon completion of the nonblocking reduction may not exactly equal the result delivered by the blocking reduction, even when specifying the same arguments in the same order. (End of advice to users.)				
10 11 12	5.12.8 No	onblocking All-Reduce				
13	MPI_IALLF	REDUCE(sendbuf, recvbuf, cou	nt, datatype,	op, comm, request)		
14 15	IN	sendbuf	starting add	lress of send buffer (choice)		
16	OUT	recvbuf	starting add	lress of receive buffer (choice)		
17 18 19	IN	count	number of e ger)	lements in send buffer (non-negative inte-		
20	IN	datatype	data type of	f elements of send buffer (handle)		
21	IN	ор	operation (h	nandle)		
22 23	IN	comm	communicat	cor (handle)		
23 24	OUT	request	communicat	tion request (handle)		
25 26 27 28 29		allreduce(const void* ser MPI_Datatype datatyp MPI_Request *request educe(sendbuf, recvbuf, c	e, MPI_Op ( )	op, MPI_Comm comm,		
30 31		ierror) BIND(C)	,			
32		*), DIMENSION(), INTENT				
33	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count					
34 35	TYPE(MPI_Datatype), INTENT(IN) :: datatype					
36	TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm					
37	TYPE(MPI_COMM), INTENI(IN) :: COMM TYPE(MPI_Request), INTENT(OUT) :: request					
38 39	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
40	MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,					
41						
42 43	<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type>					
44	This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6).					
45	1 IIIS C	an starts a nonoiocking varia		LENEDOCE (see Section 3.3.0).		
$46 \\ 47$						
48						

5.12. NONBLOCKING COLLECTIVE OPERATIONS			
5.12.9	Nonblocking Reduce-So	catter with Equal Blocks	1 2
MPI_	REDUCE_SCATTER_BLOO	CK(sendbuf, recvbuf, recvcount, datatype, op, comm, request)	3 4 5
IN	sendbuf	starting address of send buffer (choice)	6 7
00-	recvbuf	starting address of receive buffer (choice)	8
IN	recvcount	element count per block (non-negative integer)	9 10
IN	datatype	data type of elements of send and receive buffers (han- dle)	11 12
IN	ор	operation (handle)	13
IN	comm	communicator (handle)	14 15
OU	request	communication request (handle)	16
	int recvcount, MPI_Comm comm,	ck(const void* sendbuf, void* recvbuf, MPI_Datatype datatype, MPI_Op op, MPI_Request *request)	17 18 19 20 21
<pre>MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,</pre>			222 233 244 255 266 277 288 299 300 311
<pre>MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type></pre>			32 33 34 35
	'his call starts a nonblocki .10.1).	ng variant of $MPI\_REDUCE\_SCATTER\_BLOCK$ (see Sec-	36 37 38
			39 40
			41
			42
			43 44
			45
			46
			47 48
			-10

```
Nonblocking Reduce-Scatter
1
     5.12.10
\mathbf{2}
3
4
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
5
       IN
                 sendbuf
                                              starting address of send buffer (choice)
6
       OUT
\overline{7}
                 recvbuf
                                              starting address of receive buffer (choice)
8
       IN
                 recvcounts
                                              non-negative integer array specifying the number of
9
                                              elements in result distributed to each process. Array
10
                                              must be identical on all calling processes.
11
       IN
                                              data type of elements of input buffer (handle)
                 datatype
12
       IN
                                              operation (handle)
13
                 ор
14
       IN
                 comm
                                              communicator (handle)
15
       OUT
                                              communication request (handle)
                 request
16
17
     int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, const
18
                     int recvcounts[], MPI_Datatype datatype, MPI_Op op,
19
                     MPI_Comm comm, MPI_Request *request)
20
21
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
22
                     request, ierror) BIND(C)
23
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
^{24}
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
^{31}
     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
32
                     REQUEST, IERROR)
33
34
          <type> SENDBUF(*), RECVBUF(*)
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
35
36
          This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2).
37
38
39
40
41
42
43
44
45
46
47
48
```

CHAPTER 5. COLLECTIVE COMMUNICATION

# 5.12.11 Nonblocking Inclusive Scan

	N(condbut result for the	lateture en communet)
	AN(sendbuf, recvbuf, count, d	
IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative in-teger)
IN	datatype	data type of elements of input buffer (handle)
IN	ор	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
int MDT	T	
INC MPI_		uf, void* recvbuf, int count, ype, MPI_Op op, MPI_Comm comm,
	MPI_Datatype datat MPI_Request *reque	
MDT Taca	n (condbuf recubuf cou	nt, datatype, op, comm, request, ierror)
m i_isca	BIND(C)	it, datatype, op, comm, request, rerior)
TYPE		ENT(IN), ASYNCHRONOUS :: sendbuf
	(*), DIMENSION(), ASY	
INTE	GER, INTENT(IN) :: cour	nt
	(MPI_Datatype), INTENT(	
	(MPI_Op), INTENT(IN) ::	-
	(MPI_Comm), INTENT(IN)	
	(MPI_Request), INTENT(0)	-
INIE	GER, OPTIONAL, INTENT(O	UT) :: ierror
		NT, DATATYPE, OP, COMM, REQUEST, IERROR)
01	e> SENDBUF(*), RECVBUF(	
INTE	GER COUNT, DATATYPE, OP	, COMM, REQUEST, IERROR
This	call starts a nonblocking var	riant of $MPI_SCAN$ (see Section 5.11).
	0	- (

```
CHAPTER 5. COLLECTIVE COMMUNICATION
     5.12.12 Nonblocking Exclusive Scan
1
\mathbf{2}
3
4
     MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
\overline{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)
                 ор
12
13
       IN
                 comm
                                             intracommunicator (handle)
14
       OUT
                                             communication request (handle)
                 request
15
16
     int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,
17
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
18
                     MPI_Request *request)
19
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
34
         This call starts a nonblocking variant of MPI_EXSCAN (see Section 5.11.2).
35
36
     5.13
             Correctness
37
38
     A correct, portable program must invoke collective communications so that deadlock will not
39
     occur, whether collective communications are synchronizing or not. The following examples
40
     illustrate dangerous use of collective routines on intracommunicators.
41
42
     Example 5.25
43
```

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);
        break;
}
```

}

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 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.

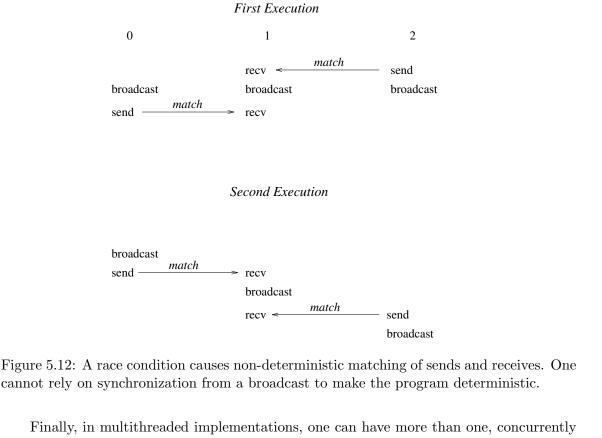
 $^{24}$ 

 $^{31}$ 

 $41 \\ 42$ 

```
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
```

<sup>45</sup> a send executed after the broadcast is received at another node before the broadcast. This
<sup>46</sup> example illustrates the fact that one should not rely on collective communication functions
<sup>47</sup> to have particular synchronization effects. A program that works correctly only when the
<sup>48</sup> first execution occurs (only when broadcast is synchronizing) is erroneous.



executing, collective communication call at a process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

- 1. All receives specify their source explicitly (no wildcards).
- 2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Then, messages belonging to successive broadcasts cannot be confused, as the order of point-to-point messages is preserved.

It is the implementor's responsibility to ensure that point-to-point messages are not confused with collective messages. One way to accomplish this is, whenever a communicator is created, to also create a "hidden communicator" for collective communication. One could achieve a similar effect more cheaply, for example, by using a hidden tag or context bit to indicate whether the communicator is used for point-to-point or collective communication. (*End of advice to implementors.*)

### 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.  $^{24}$ 

```
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_Ibarrier 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);
43
              MPI_Bcast(buf1, count, type, 0, dupcomm);
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) {
    case 0:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
        break;
}
```

 $\mathbf{2}$ 

 $^{24}$ 

```
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];
\mathbf{5}
6
\overline{7}
     switch(rank) {
8
          case 0:
            MPI_Ibarrier(comm, &reqs[0]);
9
            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
16
            MPI_Waitall(2, regs, MPI_STATUSES_IGNORE);
17
            break;
     }
18
19
          The MPI_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, &regs[1]);
^{31}
      compute(buf3);
32
     MPI_Ibcast(buf3, count, type, 0, comm, &regs[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
           Advice to implementors.
                                       The use of pipelining may generate many outstanding
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
```

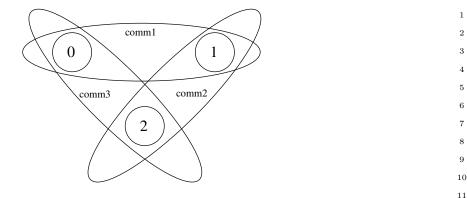


Figure 5.13: Example with overlapping communicators.

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:
      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;
}
                                                                                 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.

221

1213 14

15

16

17

18

19

2021

22 23

 $^{24}$ 25

26

28

29

30

31

32 33

35

36

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]);
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
          Finishing the second MPI_IBCAST is completely independent of the first one. This
11
     means that it is not guaranteed that the first broadcast operation is finished or even started
12
13
     after the second one is completed via reqs[1].
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

 $^{24}$ 

#### 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.

#### 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.

<sup>13</sup> <sup>14</sup> Communicators (see [21, 53, 57]) encapsulate all of these ideas in order to provide the <sup>15</sup> appropriate scope for all communication operations in MPI. Communicators are divided <sup>16</sup> into two kinds: intra-communicators for operations within a single group of processes and <sup>17</sup> inter-communicators for operations between two groups of processes.

<sup>19</sup> Caching. Communicators (see below) provide a "caching" mechanism that allows one to <sup>20</sup> associate new attributes with communicators, on par with MPI built-in features. This can <sup>21</sup> be used by advanced users to adorn communicators further, and by MPI to implement <sup>22</sup> some communicator functions. For example, the virtual-topology functions described in <sup>23</sup> 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

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.

- $45 \\ 46$
- *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 clientserver computing paradigm, where either client or server are parallel. The support of intercommunication 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 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 

 $^{24}$ 

 $^{31}$ 

ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

#### **Basic Concepts** 6.2

In this section, we turn to a more formal definition of the concepts introduced above.

#### 6.2.1 Groups

1

 $\mathbf{2}$ 

3 4

 $\mathbf{5}$ 6

> 7 8

9

20

21

22

2324

25

10 A group is an ordered set of process identifiers (henceforth processes); processes are implementation-11dependent objects. Each process in a group is associated with an integer rank. Ranks are 12contiguous and start from zero. Groups are represented by opaque group objects, and hence 13 cannot be directly transferred from one process to another. A group is used within a com-14municator to describe the participants in a communication "universe" and to rank such 15participants (thus giving them unique names within that "universe" of communication).

16There is a special pre-defined group: MPI\_GROUP\_EMPTY, which is a group with no 17members. The predefined constant MPI\_GROUP\_NULL is the value used for invalid group 18 handles. 19

- MPI\_GROUP\_EMPTY, which is a valid handle to an empty group, Advice to users. 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 26to such a table. 27
- 28Simple implementations of MPI will enumerate groups, such as in a table. However, 29 more advanced data structures make sense in order to improve scalability and memory 30 usage with large numbers of processes. Such implementations are possible with MPI.  $^{31}$ (End of advice to implementors.) 32

#### 6.2.2 Contexts

A context is a property of communicators (defined next) that allows partitioning of the 35 communication space. A message sent in one context cannot be received in another context. 36 Furthermore, where permitted, collective operations are independent of pending point-to-37 point operations. Contexts are not explicit MPI objects; they appear only as part of the 38 realization of communicators (below). 39

40 41

42

43

44

45

46

33

34

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.

47A possible implementation for a context is as a supplemental tag attached to messages 48 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 35 computation are available after MPI is initialized. For this case, MPI\_COMM\_WORLD is a 36 37 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-38 39 cally join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI\_COMM\_WORLD is a 40 41 communicator incorporating all processes with which the joining process can immediately 42communicate. Therefore, MPI\_COMM\_WORLD may simultaneously represent disjoint groups in different processes. 43

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 MPI\_COMM\_GROUP (see below). MPI does not specify the correspondence between the process rank in MPI\_COMM\_WORLD and its (machine-dependent) absolute address. Neither 48

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

does MPI specify the function of the host process, if any. Other implementation-dependent, predefined communicators may also be provided.

#### 6.3 Group Management

1

 $\mathbf{2}$ 

3 4

5 6

 $\overline{7}$ 

8

This section describes the manipulation of process groups in MPI. These operations are local and their execution does not require interprocess communication.

```
9
     6.3.1 Group Accessors
10
11
12
     MPI_GROUP_SIZE(group, size)
13
14
       IN
                                             group (handle)
                 group
15
       OUT
                                             number of processes in the group (integer)
                 size
16
17
     int MPI_Group_size(MPI_Group group, int *size)
18
19
     MPI_Group_size(group, size, ierror) BIND(C)
20
          TYPE(MPI_Group), INTENT(IN) :: group
21
          INTEGER, INTENT(OUT) ::
                                      size
22
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
^{24}
          INTEGER GROUP, SIZE, IERROR
25
26
27
     MPI_GROUP_RANK(group, rank)
28
29
       IN
                                             group (handle)
                 group
30
       OUT
                 rank
                                             rank of the calling process in group, or
^{31}
                                             MPI_UNDEFINED if the process is not a member (in-
32
                                             teger)
33
34
     int MPI_Group_rank(MPI_Group group, int *rank)
35
36
     MPI_Group_rank(group, rank, ierror) BIND(C)
37
          TYPE(MPI_Group), INTENT(IN) ::
                                              group
38
          INTEGER, INTENT(OUT) :: rank
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_GROUP_RANK(GROUP, RANK, IERROR)
41
          INTEGER GROUP, RANK, IERROR
42
43
44
45
46
47
48
```

MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2) <sup>1</sup>			
IN	group1	group1 (handle)	2 3
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)	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 8
001		MPI_UNDEFINED when no correspondence exists.	9
			10
int MP	I_Group_translate_ranks MPI_Group group2	(MPI_Group group1, int n, const int ranks1[], , int ranks2[])	11 12
MPI_Gr	oup_translate_ranks(grou BIND(C)	up1, n, ranks1, group2, ranks2, ierror)	13 14
TY	PE(MPI_Group), INTENT(IN	N) :: group1, group2	15 16
	TEGER, INTENT(IN) :: n		17
	TEGER, INTENT(OUT) :: 1 TEGER, OPTIONAL, INTENT		18
			19
		JP1, N, RANKS1, GROUP2, RANKS2, IERROR)	20 21
		(*), GROUP2, RANKS2(*), IERROR	22
	-	etermining the relative numbering of the same processes	23
		e, if one knows the ranks of certain processes in the group vant to know their ranks in a subset of that group.	24
MPL PROC NULL is a valid rank for input to MPL GROUP TRANSLATE RANKS which			
returns	$MPI\_PROC\_NULL$ as the tran	nslated rank.	26 27
			28
MPI_G	ROUP_COMPARE(group1, gr	roup2, result)	29
IN	group1	first group (handle)	30 31
IN	group2	second group (handle)	32
OUT	result	result (integer)	33
			34
int MP	I_Group_compare(MPI_Grou	<pre>up group1,MPI_Group group2, int *result)</pre>	35
MPT Cr	oun compare(groun1 grou	<pre>up2, result, ierror) BIND(C)</pre>	$\frac{36}{37}$
	PE(MPI_Group), INTENT(IN	-	38
	TEGER, INTENT(OUT) :: 1		39
IN	TEGER, OPTIONAL, INTENT	(OUT) :: ierror	40
MPI_GR	OUP_COMPARE(GROUP1, GROU	JP2, RESULT, IERROR)	41
	TEGER GROUP1, GROUP2, RI		42 43
MPI ID	ENT results if the group mem	bers and group order is exactly the same in both groups.	43
		and group2 are the same handle. MPI_SIMILAR results if	45
		t the order is different. MPI_UNEQUAL results otherwise.	46
			47
			48

```
6.3.2 Group Constructors
```

 $\mathbf{2}$ Group constructors are used to subset and superset existing groups. These constructors 3 construct new groups from existing groups. These are local operations, and distinct groups 4 may be defined on different processes; a process may also define a group that does not 5include itself. Consistent definitions are required when groups are used as arguments in 6 communicator-building functions. MPI does not provide a mechanism to build a group 7 from scratch, but only from other, previously defined groups. The base group, upon which 8 all other groups are defined, is the group associated with the initial communicator 9 MPI\_COMM\_WORLD (accessible through the function MPI\_COMM\_GROUP). 10

Rationale. In what follows, there is no group duplication function analogous to MPI\_COMM\_DUP, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

Advice to implementors. Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (End of advice to implementors.)

```
22
23
```

 $^{24}$ 

1

11

12

13

14

15

16 17

18

19

20

21

MPI\_COMM\_GROUP(comm, group)

25IN comm communicator (handle) 26OUT group group corresponding to comm (handle) 2728int MPI\_Comm\_group(MPI\_Comm comm, MPI\_Group \*group) 29 30 MPI\_Comm\_group(comm, group, ierror) BIND(C)  $^{31}$ TYPE(MPI\_Comm), INTENT(IN) :: comm 32 TYPE(MPI\_Group), INTENT(OUT) :: group 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34MPI\_COMM\_GROUP(COMM, GROUP, IERROR) 35 INTEGER COMM, GROUP, IERROR 36 37 MPI\_COMM\_GROUP returns in group a handle to the group of comm. 38 39 40MPI\_GROUP\_UNION(group1, group2, newgroup) 41 IN first group (handle) group1 42IN group2 second group (handle) 43 44OUT newgroup union group (handle) 4546int MPI\_Group\_union(MPI\_Group group1, MPI\_Group group2, 47MPI\_Group \*newgroup) 48

MPI_Grou	up_union(group1, g	group2, newgroup, ierror) BIND(C)	1
		ENT(IN) :: group1, group2	2
	-	ENT(OUT) :: newgroup	3
INTE	EGER, OPTIONAL, IN	NTENT(OUT) :: ierror	4
MPI_GROU	JP_UNION(GROUP1, (	GROUP2, NEWGROUP, IERROR)	5 6
INTE	EGER GROUP1, GROUP	P2, NEWGROUP, IERROR	7
			8
			9
MPI_GR0	OUP_INTERSECTIO	N(group1, group2, newgroup)	10
IN	group1	first group (handle)	11
IN	group2	second group (handle)	12
OUT	newgroup	intersection group (handle)	13 14
		monoconon Broap (namato)	14
int MPI	Group intersection	on(MPI_Group group1, MPI_Group group2,	16
	MPI_Group *:		17
NDT O	-	· ·	18
		roup1, group2, newgroup, ierror) BIND(C) ENT(IN) :: group1, group2	19
		ENT(OUT) :: newgroup	20
	-	NTENT(OUT) :: ierror	21
			22
		ROUP1, GROUP2, NEWGROUP, IERROR)	23 24
INTE	GER GRUUP1, GRUUP	P2, NEWGROUP, IERROR	24 25
			26
			27
		group1, group2, newgroup)	28
IN	group1	first group (handle)	29
IN	group2	second group (handle)	30
OUT	newgroup	difference group (handle)	31
			32
int MPI_	Group_difference	(MPI_Group group1, MPI_Group group2,	33
	MPI_Group *	newgroup)	34 35
MPT Grou	n difference(grou	<pre>up1, group2, newgroup, ierror) BIND(C)</pre>	36
		ENT(IN) :: group1, group2	37
	-	ENT(OUT) :: newgroup	38
	EGER, OPTIONAL, IN		39
MPT CROI		JP1, GROUP2, NEWGROUP, IERROR)	40
		P2, NEWGROUP, IERROR	41
			42
The set-l	ike operations are de	inned as iollows:	43 44
		rst group (group1), followed by all elements of second group	44
(gro	pup2) not in the first	group.	46
intersec	$\mathbf{t}$ all elements of the	e first group that are also in the second group, ordered as in	47
	first group.	G F F F F F F F F F F F F F F F F F F F	48

1 difference all elements of the first group that are not in the second group, ordered as in  $\mathbf{2}$ the first group. 3 Note that for these operations the order of processes in the output group is determined 4 primarily by order in the first group (if possible) and then, if necessary, by order in the 5second group. Neither union nor intersection are commutative, but both are associative. 6 The new group can be empty, that is, equal to MPI\_GROUP\_EMPTY. 7 8 9 MPI\_GROUP\_INCL(group, n, ranks, newgroup) 10 IN group (handle) group 11 12IN number of elements in array ranks (and size of n 13 newgroup) (integer) 14IN ranks of processes in group to appear in ranks 15newgroup (array of integers) 16OUT newgroup new group derived from above, in the order defined by 17ranks (handle) 18 19int MPI\_Group\_incl(MPI\_Group group, int n, const int ranks[], 2021MPI\_Group \*newgroup) 22MPI\_Group\_incl(group, n, ranks, newgroup, ierror) BIND(C) 23TYPE(MPI\_Group), INTENT(IN) :: group  $^{24}$ INTEGER, INTENT(IN) :: n, ranks(n) 25TYPE(MPI\_Group), INTENT(OUT) :: newgroup 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 27MPI\_GROUP\_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) 28INTEGER GROUP, N, RANKS(\*), NEWGROUP, IERROR 2930 The function MPI\_GROUP\_INCL creates a group newgroup that consists of the  $^{31}$ n processes in group with ranks  $ranks[0], \ldots, ranks[n-1]$ ; the process with rank i in newgroup 32 is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid 33 rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, 34 then newgroup is MPI\_GROUP\_EMPTY. This function can, for instance, be used to reorder 35 the elements of a group. See also MPI\_GROUP\_COMPARE. 36 37 38MPI\_GROUP\_EXCL(group, n, ranks, newgroup) 39 IN group (handle) group 40IN number of elements in array ranks (integer) 41 n 42IN ranks array of integer ranks in group not to appear in 43 newgroup 44OUT newgroup new group derived from above, preserving the order 45defined by group (handle) 46 47 48

<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>			$\frac{1}{2}$
-	<pre>MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C)     TYPE(MPI_Group), INTENT(IN) :: group</pre>		
	GER, INTENT(IN) :: n, rai		5 6
	(MPI_Group), INTENT(OUT)	<b>o i</b>	7
INTEC	SER, OPTIONAL, INTENT(OUT)	) :: ierror	8
	P_EXCL(GROUP, N, RANKS, NI		9
INTEC	GER GROUP, N, RANKS(*), NI	EWGROUP, IERROR	10 11
by deleting	g from <b>group</b> those processes	eates a group of processes <b>newgroup</b> that is obtained with ranks ranks[0] , ranks[n-1]. The ordering of	12 13
		ordering in group. Each of the n elements of ranks	14
	If $n = 0$ , then newgroup is in	ements must be distinct; otherwise, the program is dentical to group	15
cironeous.			16 17
		<b>`</b>	18
	UP_RANGE_INCL(group, n, ra	,	19
IN	group	group (handle)	20
IN	n	number of triplets in array ranges (integer)	21
IN	ranges	a one-dimensional array of integer triplets, of the form	22
		(first rank, last rank, stride) indicating ranks in $group$	23 24
		of processes to be included in newgroup	24
OUT	newgroup	new group derived from above, in the order defined by	26
		ranges (handle)	27
			28
int MPI_(		p group, int n, int ranges[][3],	29
	MPI_Group *newgroup)		30 31
	<pre>MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)</pre>		
	TYPE(MPI_Group), INTENT(IN) :: group		
	INTEGER, INTENT(IN) :: n, ranges(3,n) TYPE(MPI_Group), INTENT(OUT) :: newgroup		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror		35	
			36 37
	MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR		38
			39
If ranges c	If ranges consists of the triplets		40
(fir.	$(first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)$		41
then <b>new</b> o	then newgroup consists of the sequence of processes in group with ranks		42
_			$43 \\ 44$
firs	$t_1, first_1 + stride_1, \dots, first_1$	$+ \left  \frac{last_1 - first_1}{stride_1} \right  stride_1, \dots,$	45
v	, , <b>,</b> –	$\lfloor striae_1 \rfloor$	46
fina	t first _ stride first	$\left  \frac{last_n - first_n}{last_n - first_n} \right _{stride}$	47
jurs	$first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$		

1 Each computed rank must be a valid rank in group and all computed ranks must be  $\mathbf{2}$ distinct, or else the program is erroneous. Note that we may have  $first_i > last_i$ , and  $stride_i$ 3 may be negative, but cannot be zero.

4 The functionality of this routine is specified to be equivalent to expanding the array  $\mathbf{5}$ of ranges to an array of the included ranks and passing the resulting array of ranks and 6 other arguments to MPI\_GROUP\_INCL. A call to MPI\_GROUP\_INCL is equivalent to a call  $\overline{7}$ to MPI\_GROUP\_RANGE\_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the 8 argument ranges.

```
MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)
```

9 10

11

29 30

 $^{31}$ 

32

40

45

```
12
       IN
                                               group (handle)
                 group
13
       IN
                 n
                                               number of elements in array ranges (integer)
14
                                              a one-dimensional array of integer triplets of the form
       IN
                 ranges
15
                                               (first rank, last rank, stride), indicating the ranks in
16
                                               group of processes to be excluded from the output
17
                                               group newgroup.
18
19
        OUT
                  newgroup
                                              new group derived from above, preserving the order
20
                                              in group (handle)
21
22
      int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
23
                     MPI_Group *newgroup)
^{24}
     MPI_Group_range_excl(group, n, ranges, newgroup, ierror) BIND(C)
25
          TYPE(MPI_Group), INTENT(IN) :: group
26
          INTEGER, INTENT(IN) :: n, ranges(3,n)
27
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
```

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

Each computed rank must be a valid rank in group and all computed ranks must be distinct, 33 34or else the program is erroneous.

ierror

The functionality of this routine is specified to be equivalent to expanding the array of 35 ranges to an array of the excluded ranks and passing the resulting array of ranks and other 36 arguments to MPI\_GROUP\_EXCL. A call to MPI\_GROUP\_EXCL is equivalent to a call to 37 MPI\_GROUP\_RANGE\_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the 38 argument ranges. 39

```
The range operations do not explicitly enumerate ranks, and
           Advice to users.
41
           therefore are more scalable if implemented efficiently. Hence, we recommend MPI
42
           programmers to use them whenenever possible, as high-quality implementations will
43
           take advantage of this fact. (End of advice to users.)
44
```

```
Advice to implementors. The range operations should be implemented, if possible,
46
           without enumerating the group members, in order to obtain better scalability (time
47
           and space). (End of advice to implementors.)
48
```

6.3.3 Group Destructors

0.5.5 Group Destructors	
MPI_GROUP_FREE(group) INOUT group	group (handle)
int MPI_Group_free(MPI_Group *grou	p)
<pre>MPI_Group_free(group, ierror) BIND TYPE(MPI_Group), INTENT(INOUT) INTEGER, OPTIONAL, INTENT(OUT)</pre>	:: group
MPI_GROUP_FREE(GROUP, IERROR) INTEGER GROUP, IERROR	

This operation marks a group object for deallocation. The handle group is set to MPI\_GROUP\_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI\_COMM\_GROUP, MPI\_COMM\_CREATE, MPI\_COMM\_DUP, and MPI\_COMM\_IDUP, and decremented for each call to MPI\_GROUP\_FREE or MPI\_COMM\_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

## 6.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (End of advice to implementors.)

6.4.1 Communicator Accessors			37
0.1.1 00			38
The follow	ing are all local operations.		39
			40
			41
MPI_COM	M_SIZE(comm, size)		42
IN	comm	communicator (handle)	43
OUT	size	number of processes in the group of comm (integer)	44
001	5120	number of processes in the group of comm (moeger)	45
			46
<pre>int MPI_Comm_size(MPI_Comm comm, int *size)</pre>		47	
MPI_Comm_size(comm, size, ierror) BIND(C)		48	

1	TYPE(MPI_Comm), INTE	NT(IN) :: comm	
2	INTEGER, INTENT(OUT) :: size		
3	INTEGER, OPTIONAL, I	NTENT(OUT) :: ierror	
4	MPI_COMM_SIZE(COMM, SIZE	TERROR)	
5	INTEGER COMM, SIZE,	-	
6			
7	Detional This from the	···· ·································	
8 9		ion is equivalent to accessing the communicator's group with see above), computing the size using MPI_GROUP_SIZE, and	
10		rary group via MPI_GROUP_FREE. However, this function is	
11		this shortcut was introduced. ( <i>End of rationale.</i> )	
12		······································	
13	Advice to users. Th	is function indicates the number of processes involved in a	
14		PI_COMM_WORLD, it indicates the total number of processes	
15		mber of processes has been changed by using the functions	
16	-	0; note that the number of processes in MPI_COMM_WORLD	
17	0 0	the life of an MPI program.	
18		with the next call to determine the amount of concurrency	
19	-	library or program. The following call, MPI_COMM_RANK	
20 21		e process that calls it in the range from $0 \dots \text{size} -1$ , where size	
21	is the return value of W	IPI_COMM_SIZE.(End of advice to users.)	
23			
24			
25	MPI_COMM_RANK(comm, ra	ank)	
26	IN comm	communicator (handle)	
27	OUT rank	rank of the calling process in group of $comm$ (integer)	
28 29			
30	int MPI_Comm_rank(MPI_Com	nm comm, int *rank)	
31	MPI_Comm_rank(comm, rank	, ierror) BIND(C)	
32	TYPE(MPI_Comm), INTE		
33	INTEGER, INTENT(OUT)		
34	INTEGER, OPTIONAL, I	NTENT(OUT) :: ierror	
35 36	MPI_COMM_RANK(COMM, RANK	, IERROR)	
30 37	INTEGER COMM, RANK,		
38			
39	Rationale. This funct	ion is equivalent to accessing the communicator's group with	
40		see above), computing the rank using MPI_GROUP_RANK,	
41		nporary group via MPI_GROUP_FREE. However, this function	
42	is so commonly used th	at this shortcut was introduced. (End of rationale.)	
43			
44		unction gives the rank of the process in the particular commu-	
45	С .	eful, as noted above, in conjunction with MPI_COMM_SIZE.	
46	Many programs will be	written with the master-slave model, where one process (such	
47	, 1 1	·	
48	-	ss) will play a supervisory role, and the other processes will s. In this framework, the two preceding calls are useful for	

determining the roles of the various processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)				
IN	comm1	first communicator (handle)		
IN	comm2	second communicator (handle)		
OUT	result	result (integer)		
<pre>int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result) MPI_Comm_compare(comm1, comm2, result, ierror) BIND(C)     TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2     INTEGER, INTENT(OUT) :: result     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) INTEGER COMM1, COMM2, RESULT, IERROR				

MPI\_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI\_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI\_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI\_UNEQUAL results otherwise.

#### 6.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm, with the exception of MPI\_COMM\_CREATE\_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.

*Rationale.* Note that there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. The base communicator for all MPI communicators is predefined outside of MPI, and is MPI\_COMM\_WORLD. This model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

This chapter presents the following communicator construction routines: MPI\_COMM\_CREATE, MPI\_COMM\_DUP, MPI\_COMM\_IDUP, MPI\_COMM\_DUP\_WITH\_INFO, and MPI\_COMM\_SPLIT can be used to create both intracommunicators and intercommunicators; MPI\_COMM\_CREATE\_GROUP and MPI\_INTERCOMM\_MERGE (see Section 6.6.2) can be used to create intracommunicators; and MPI\_INTERCOMM\_CREATE (see Section 6.6.2) can be used to create intercommunicators.

An intracommunicator involves a single group while an intercommunicator involves 45 two groups. Where the following discussions address intercommunicator semantics, the 46 two groups in an intercommunicator are called the *left* and *right* groups. A process in an 47 intercommunicator is a member of either the left or the right group. From the point of view 48

 $^{24}$ 

<sup>1</sup> of that process, the group that the process is a member of is called the *local* group; the <sup>2</sup> other group (relative to that process) is the *remote* group. The left and right group labels <sup>3</sup> give us a way to describe the two groups in an intercommunicator that is not relative to <sup>4</sup> any particular process (as the local and remote groups are).

```
<sup>6</sup>
7 MPI_COMM_DUP(comm, newcomm)
```

5

```
8
       IN
                                          communicator (handle)
                comm
9
       OUT
                                          copy of comm (handle)
                newcomm
10
11
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
12
13
    MPI_Comm_dup(comm, newcomm, ierror) BIND(C)
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
16
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                               ierror
17
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
18
         INTEGER COMM, NEWCOMM, IERROR
19
```

<sup>20</sup> MPI\_COMM\_DUP duplicates the existing communicator comm with associated key <sup>21</sup> values, topology information, and info hints. For each key value, the respective copy callback <sup>22</sup> function determines the attribute value associated with this key in the new communicator; <sup>23</sup> one particular action that a copy callback may take is to delete the attribute from the new <sup>24</sup> communicator. Returns in newcomm a new communicator with the same group or groups, <sup>25</sup> same topology, same info hints, any copied cached information, but a new context (see <sup>26</sup> Section 6.7.1).

Advice to users. This operation is used to provide a parallel library with a duplicate 28communication space that has the same properties as the original communicator. This 29 includes any attributes (see below), topologies (see Chapter 7), and associated info 30 hints (see Section 6.4.4). This call is valid even if there are pending point-to-point 31communications involving the communicator comm. A typical call might involve a 32 MPI\_COMM\_DUP at the beginning of the parallel call, and an MPI\_COMM\_FREE of 33 that duplicated communicator at the end of the call. Other models of communicator 34 management are also possible. 35

This call applies to both intra- and inter-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.)

40 41 42

43

36

37

38

39

27

MPI\_COMM\_DUP\_WITH\_INFO(comm, info, newcomm)

44	IN	comm	communicator (handle)
45 46	IN	info	info object (handle)
40 47	OUT	newcomm	copy of comm (handle)
48			

```
1
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
                                                                                       2
MPI_Comm_dup_with_info(comm, info, newcomm, ierror) BIND(C)
                                                                                        3
    TYPE(MPI_Comm), INTENT(IN) ::
                                      comm
                                                                                        4
    TYPE(MPI_Info), INTENT(IN) ::
                                      info
                                                                                        5
    TYPE(MPI_Comm), INTENT(OUT) ::
                                       newcomm
                                                                                        6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
                                                                                       9
    INTEGER COMM, INFO, NEWCOMM, IERROR
                                                                                       10
    MPI_COMM_DUP_WITH_INFO behaves exactly as MPI_COMM_DUP except that the
                                                                                       11
info hints associated with the communicator comm are not duplicated in newcomm. The
                                                                                       12
hints provided by the argument info are associated with the output communicator newcomm
                                                                                       13
instead.
                                                                                       14
                                                                                       15
     Rationale. It is expected that some hints will only be valid at communicator creation
                                                                                       16
     time. However, for legacy reasons, most communicator creation calls do not provide
                                                                                       17
     an info argument. One may associate info hints with a duplicate of any communicator
                                                                                       18
     at creation time through a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)
                                                                                       19
                                                                                       20
                                                                                       21
MPI_COMM_IDUP(comm, newcomm, request)
                                                                                       22
                                                                                       23
  IN
                                      communicator (handle)
           comm
                                                                                       ^{24}
  OUT
           newcomm
                                      copy of comm (handle)
                                                                                       25
                                                                                       26
                                      communication request (handle)
  OUT
           request
                                                                                       27
                                                                                       28
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
                                                                                       29
MPI_Comm_idup(comm, newcomm, request, ierror) BIND(C)
                                                                                       30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                       31
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                       32
    TYPE(MPI_Request), INTENT(OUT) ::
                                           request
                                                                                       33
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
                                                                                       34
```

MPI\_COMM\_IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR

MPI\_COMM\_IDUP is a nonblocking variant of MPI\_COMM\_DUP. The semantics of MPI\_COMM\_IDUP are as if MPI\_COMM\_DUP was executed at the time that MPI\_COMM\_IDUP is called. For example, attributes changed after MPI\_COMM\_IDUP will not be copied to the new communicator. All restrictions and assumptions for nonblocking collective operations (see Section 5.12) apply to MPI\_COMM\_IDUP and the returned request.

It is erroneous to use the communicator **newcomm** as an input argument to other MPI functions before the MPI\_COMM\_IDUP operation completes.

*Rationale.* This functionality is crucial for the development of purely nonblocking libraries (see [36]). (*End of rationale.*)

35

36

37

38

39

40

41

42

43

44

45 46

47

240 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI\_COMM\_CREATE(comm, group, newcomm)

1

2 IN communicator (handle) comm 3 IN group, which is a subset of the group of comm (handle) group 4 5OUT newcomm new communicator (handle) 6  $\overline{7}$ int MPI\_Comm\_create(MPI\_Comm comm, MPI\_Group group, MPI\_Comm \*newcomm) 8 MPI\_Comm\_create(comm, group, newcomm, ierror) BIND(C) 9 TYPE(MPI\_Comm), INTENT(IN) :: comm 10 TYPE(MPI\_Group), INTENT(IN) :: group 11 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14MPI\_COMM\_CREATE(COMM, GROUP, NEWCOMM, IERROR) 15INTEGER COMM, GROUP, NEWCOMM, IERROR 1617If comm is an intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information 18 propagates from comm to newcomm. Each process must call MPI\_COMM\_CREATE with 19a group argument that is a subgroup of the group associated with comm; this could be 2021MPI\_GROUP\_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the 22function with the same group as argument, that is the same processes in the same order. 23Otherwise, the call is erroneous. This implies that the set of groups specified across the  $^{24}$ processes must be disjoint. If the calling process is a member of the group given as group 2526argument, then **newcomm** is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI\_GROUP\_EMPTY, 27then MPI\_COMM\_NULL is returned as newcomm. The function is collective and must be 28called by all processes in the group of comm. 2930 Rationale. The interface supports the original mechanism from MPI-1.1, which re-31quired the same group in all processes of comm. It was extended in MPI-2.2 to allow 32 the use of disjoint subgroups in order to allow implementations to eliminate unnec-33 essary communication that MPI\_COMM\_SPLIT would incur when the user already 34 knows the membership of the disjoint subgroups. (End of rationale.) 3536 The requirement that the entire group of comm participate in the call Rationale. 37 stems from the following considerations: 38 39 • It allows the implementation to layer MPI\_COMM\_CREATE on top of regular 40 collective communications. 41 • It provides additional safety, in particular in the case where partially overlapping 42groups are used to create new communicators. 43 • It permits implementations to sometimes avoid communication related to context 44creation. 4546(End of rationale.) 47 48

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) to further 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 must 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.

```
38
MPI_Comm inter_comm, new_inter_comm;
                                                                            39
MPI_Group local_group, group;
          rank = 0; /* rank on left side to include in
                                                                            40
int
                                                                            41
                        new inter-comm */
                                                                            42
/* Construct the original intercommunicator: "inter_comm" */
                                                                            43
                                                                            44
. . .
                                                                            45
                                                                            46
/* Construct the group of processes to be in new
                                                                            47
   intercommunicator */
                                                                            48
if (/* I'm on the left side of the intercommunicator */) {
```

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

```
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
                                  T
13
                                 1
14
                               081
15
                                                                   1
16
17
                                                                       Ж
                                                                  2
18
19
20
21
22
     Figure 6.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercom-
23
     municators. The input groups are those in the grey circle.
^{24}
25
                 MPI_Comm_group ( inter_comm, &local_group );
26
                 MPI_Group_incl ( local_group, 1, &rank, &group );
27
                 MPI_Group_free ( &local_group );
28
              }
29
              else
30
                 MPI_Comm_group ( inter_comm, &group );
^{31}
32
              MPI_Comm_create ( inter_comm, group, &new_inter_comm );
33
              MPI_Group_free( &group );
34
35
36
     MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)
37
38
       IN
                 comm
                                             intracommunicator (handle)
39
       IN
                                             group, which is a subset of the group of comm (handle)
                 group
40
41
       IN
                                             tag (integer)
                 tag
42
       OUT
                                             new communicator (handle)
                 newcomm
43
44
     int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
45
                     MPI_Comm *newcomm)
46
47
     MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
48
          TYPE(MPI_Comm), INTENT(IN) :: comm
```

1 TYPE(MPI\_Group), INTENT(IN) :: group 2 INTEGER, INTENT(IN) :: tag 3 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 MPI\_COMM\_CREATE\_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) 6 INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR 7 8 MPI\_COMM\_CREATE\_GROUP is similar to MPI\_COMM\_CREATE; however, 9 MPI\_COMM\_CREATE must be called by all processes in the group of 10 comm, whereas MPI\_COMM\_CREATE\_GROUP must be called by all processes in group, 11 which is a subgroup of the group of comm. In addition, MPI\_COMM\_CREATE\_GROUP requires that comm is an intracommunicator. MPI\_COMM\_CREATE\_GROUP returns a new 1213 intracommunicator, newcomm, for which the group argument defines the communication 14group. No cached information propagates from comm to newcomm. Each process must 15provide a group argument that is a subgroup of the group associated with comm; this 16could be MPI\_GROUP\_EMPTY. If a non-empty group is specified, then all processes in that 17 group must call the function, and each of these processes must provide the same arguments, 18including a group that contains the same members with the same ordering. Otherwise 19the call is erroneous. If the calling process is a member of the group given as the group 20argument, then newcomm is a communicator with group as its associated group. If the 21calling process is not a member of group, e.g., group is MPI\_GROUP\_EMPTY, then the call 22is a local operation and MPI\_COMM\_NULL is returned as newcomm. 23Rationale. Functionality similar to MPI\_COMM\_CREATE\_GROUP can be imple- $^{24}$ mented through repeated MPI\_INTERCOMM\_CREATE and 25MPI\_INTERCOMM\_MERGE calls that start with the MPI\_COMM\_SELF communica-26tors at each process in group and build up an intracommunicator with group 27group [16]. Such an algorithm requires the creation of many intermediate communi-28cators; MPI\_COMM\_CREATE\_GROUP can provide a more efficient implementation 29that avoids this overhead. (End of rationale.) 30 31Advice to users. An intercommunicator can be created collectively over processes in 32 the union of the local and remote groups by creating the local communicator using 33 MPI\_COMM\_CREATE\_GROUP and using that communicator as the local communi-34 cator argument to MPI\_INTERCOMM\_CREATE. (End of advice to users.) 3536 The tag argument does not conflict with tags used in point-to-point communication and 37 is not permitted to be a wildcard. If multiple threads at a given process perform concurrent 38 MPI\_COMM\_CREATE\_GROUP operations, the user must distinguish these operations by 39 providing different tag or comm arguments. 40 41 Advice to users. MPI\_COMM\_CREATE may provide lower overhead than 42MPI\_COMM\_CREATE\_GROUP because it can take advantage of collective communi-43 cation on comm when constructing newcomm. (End of advice to users.) 4445464748

MPI\_COMM\_SPLIT(comm, color, key, newcomm)

1

_	-	- (				
2 3	IN	comm	communicator (handle)			
4	IN	color	control of subset assignment (integer)			
5	IN	key	control of rank assignment (integer)			
6 7	OUT	newcomm	new communicator (handle)			
8 9	int MPI_	Comm_split(MPI	[_Comm comm, int color, int key, MPI_Comm *newcomm)			
10	MPI_Comm	_split(comm, c	color, key, newcomm, ierror) BIND(C)			
11	TYPE	C(MPI_Comm), IN	NTENT(IN) :: comm			
12	INTE	GER, INTENT(IN	N) :: color, key			
13	TYPE	C(MPI_Comm), IN	NTENT(OUT) :: newcomm			
14	INTE	GER, OPTIONAL,	, INTENT(OUT) :: ierror			
15	МРТ СОММ		TOLOR KEV NEUCOMM TERROR)			
16		MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR				
17		GER COMM, COLC	JR, KEI, NEWGOFFF, TERROR			
18		-	he group associated with <b>comm</b> into disjoint subgroups, one for			
19 20	each value of color. Each subgroup contains all processes of the same color. Within each					
20	subgroup, the processes are ranked in the order defined by the value of the argument					
22	key, with ties broken according to their rank in the old group. A new communicator is					
23	created for each subgroup and returned in newcomm. A process may supply the color value					
23	MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective					
24	call, but each process is permitted to provide different values for color and key.					
26			nicator comm, a call to MPI_COMM_CREATE(comm, group, new-			
27	,	*	call to MPI_COMM_SPLIT(comm, color, key, newcomm), where			
28	-		rs of their group argument provide $color =$ number of the group			
29	(based on a unique numbering of all disjoint groups) and key = rank in group, and all					
30	-		nbers of their group argument provide $color = MPI_UNDEFINED$ .			
31	The	value of color mu	st be non-negative or MPI_UNDEFINED.			
32	Ada	vice to users.	This is an extremely powerful mechanism for dividing a single			
33			up of processes into $k$ subgroups, with $k$ chosen implicitly by the			
00		00	r = f = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1			

user (by the number of colors asserted over all the processes). Each resulting com-34municator will be non-overlapping. Such a division could be useful for defining a 35 hierarchy of computations, such as for multigrid, or linear algebra. For intracommu-36 nicators, MPI\_COMM\_SPLIT provides similar capability as MPI\_COMM\_CREATE to 37 split a communicating group into disjoint subgroups. MPI\_COMM\_SPLIT is useful 38 when some processes do not have complete information of the other members in their 39 group, but all processes know (the color of) the group to which they belong. In this 40 case, the MPI implementation discovers the other group members via communication. 41 MPI\_COMM\_CREATE is useful when all processes have complete information of the 42members of their group. In this case, MPI can avoid the extra communication required 43 to discover group membership. MPI\_COMM\_CREATE\_GROUP is useful when all pro-44cesses in a given group have complete information of the members of their group and 45synchronization with processes outside the group can be avoided. 46

<sup>47</sup> Multiple calls to MPI\_COMM\_SPLIT can be used to overcome the requirement that <sup>48</sup> any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created. Creative use of the color and key in such splitting operations is encouraged.

Note that, for a fixed color, the keys need not be unique. It is MPI\_COMM\_SPLIT's responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.

Essentially, making the key value zero for all processes of a given color means that one does not really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)

*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.

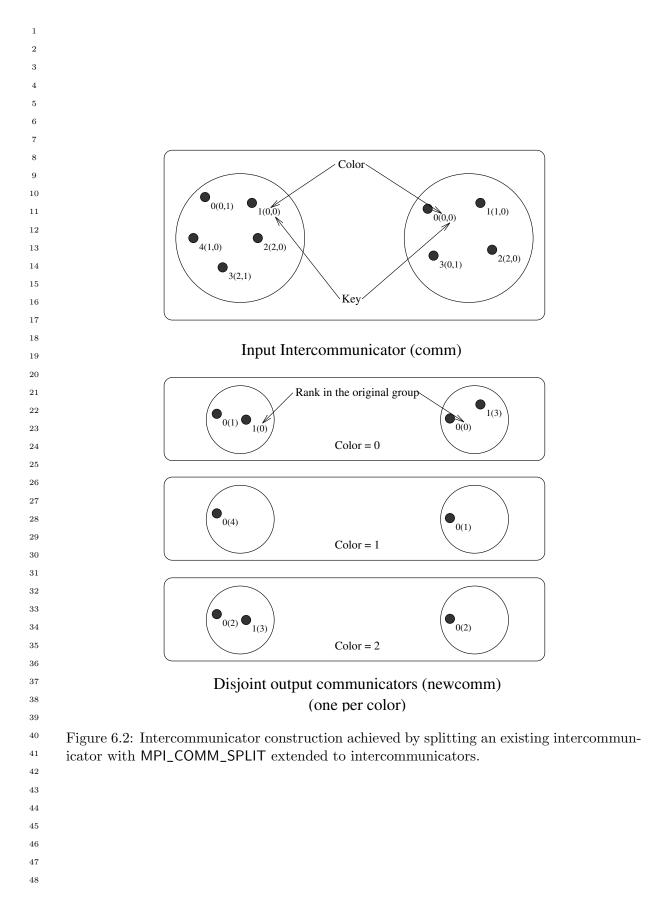
Advice to users. For intercommunicators, MPI\_COMM\_SPLIT is more general than MPI\_COMM\_CREATE. A single call to MPI\_COMM\_SPLIT can create a set of disjoint intercommunicators, while a call to MPI\_COMM\_CREATE creates only one. (*End of advice to users.*)

**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.

```
30
/* Client code */
                                                                            31
MPI_Comm multiple_server_comm;
                                                                            32
MPI_Comm single_server_comm;
                                                                            33
int
          color, rank, num_servers;
                                                                            34
/* Create intercommunicator with clients and servers:
                                                                            35
                                                                            36
   multiple_server_comm */
                                                                            37
. . .
                                                                            38
                                                                            39
/* Find out the number of servers available */
MPI_Comm_remote_size ( multiple_server_comm, &num_servers );
                                                                            40
                                                                            41
                                                                            42
/* Determine my color */
MPI_Comm_rank ( multiple_server_comm, &rank );
                                                                            43
                                                                            44
color = rank % num_servers;
                                                                            45
                                                                            46
/* Split the intercommunicator */
                                                                            47
MPI_Comm_split ( multiple_server_comm, color, rank,
                                                                            48
                  &single_server_comm );
```

 $\overline{7}$ 

 $^{24}$ 



The following is the corresponding server code:

```
\mathbf{2}
         /* Server code */
                                                                                        3
         MPI_Comm multiple_client_comm;
                                                                                        4
         MPI_Comm single_server_comm;
                                                                                        5
         int
                    rank;
                                                                                        6
                                                                                        7
         /* Create intercommunicator with clients and servers:
            multiple_client_comm */
                                                                                        9
         . . .
                                                                                       10
                                                                                       11
         /* Split the intercommunicator for a single server per group
                                                                                       12
            of clients */
                                                                                       13
         MPI_Comm_rank ( multiple_client_comm, &rank );
                                                                                       14
         MPI_Comm_split ( multiple_client_comm, rank, 0,
                                                                                       15
                            &single_server_comm );
                                                                                       16
                                                                                       17
                                                                                       18
MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)
                                                                                       19
                                                                                       20
  IN
                                      communicator (handle)
           comm
                                                                                       21
  IN
           split_type
                                      type of processes to be grouped together (integer)
                                                                                       22
                                                                                       23
  IN
           key
                                      control of rank assignment (integer)
                                                                                       ^{24}
  IN
           info
                                      info argument (handle)
                                                                                       25
  OUT
           newcomm
                                      new communicator (handle)
                                                                                       26
                                                                                       27
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
                                                                                       28
              MPI_Info info, MPI_Comm *newcomm)
                                                                                       29
                                                                                       30
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) BIND(C)
                                                                                       31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                       32
    INTEGER, INTENT(IN) :: split_type, key
                                                                                       33
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                       34
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                       35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       36
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
                                                                                       37
    INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
                                                                                       38
```

This function partitions the group associated with comm into disjoint subgroups, based on the type specified by split\_type. Each subgroup contains all processes of the same type. 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. This is a collective call; all processes must provide the same split\_type, but each process is permitted to provide different values for key. An exception to this rule is that a process may supply the type value MPI\_UNDEFINED, in which case newcomm returns MPI\_COMM\_NULL.

The following type is predefined by MPI:

1

39

40

41

42

43

44

45

46

47

1 MPI\_COMM\_TYPE\_SHARED — this type splits the communicator into subcommunicators,  $\mathbf{2}$ each of which can create a shared memory region. 3 4 Advice to implementors. Implementations can define their own types, or use the info argument, to assist in creating communicators that help expose platform-specific 5information to the application. (End of advice to implementors.) 6  $\overline{7}$ 8 6.4.3 Communicator Destructors 9 10 11MPI\_COMM\_FREE(comm) 12INOUT communicator to be destroyed (handle) comm 13 14int MPI\_Comm\_free(MPI\_Comm \*comm) 1516MPI\_Comm\_free(comm, ierror) BIND(C) 17TYPE(MPI\_Comm), INTENT(INOUT) :: comm 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19MPI\_COMM\_FREE(COMM, IERROR) 20INTEGER COMM, IERROR 2122This collective operation marks the communication object for deallocation. The handle 23is set to MPI\_COMM\_NULL. Any pending operations that use this communicator will com- $^{24}$ 

<sup>23</sup> is set to MPI\_COMM\_NULL. Any pending operations that use this communicator will com-<sup>24</sup> plete normally; the object is actually deallocated only if there are no other active references <sup>26</sup> to it. This call applies to intra- and inter-communicators. The delete callback functions for <sup>27</sup> all cached attributes (see Section 6.7) are called in arbitrary order.

27 28

29

30

31

32

33

34

35 36

37

Advice to implementors. A reference-count mechanism may be used: the reference count is incremented by each call to MPI\_COMM\_DUP or MPI\_COMM\_IDUP, and decremented by each call to MPI\_COMM\_FREE. The object is ultimately deallocated when the count reaches 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.*)

#### 6.4.4 Communicator Info

 <sup>38</sup>Hints specified via info (see Chapter 9) allow a user to provide information to direct opti-<sup>39</sup>mization. Providing hints may enable an implementation to deliver increased performance <sup>40</sup>or minimize use of system resources. However, hints do not change the semantics of any MPI <sup>41</sup>interfaces. In other words, an implementation is free to ignore all hints. Hints are specified <sup>42</sup>on a per communicator basis, in MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO, <sup>43</sup>MPI\_COMM\_SPLIT\_TYPE, MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, and

<sup>44</sup> MPI\_DIST\_GRAPH\_CREATE, via the opaque info object. When an info object that speci-<sup>45</sup> fies a subset of valid hints is passed to MPI\_COMM\_SET\_INFO, there will be no effect on <sup>46</sup> 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.*)

Info hints are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI\_COMM\_DUP or MPI\_COMM\_IDUP. In this case, all hints associated with the original communicator are also applied to the duplicated communicator.

MPI\_COMM\_SET\_INFO(comm, info)

INOUT	comm	communicator (handle)
IN	info	info object (handle)

int MPI\_Comm\_set\_info(MPI\_Comm comm, MPI\_Info info)

- MPI\_Comm\_set\_info(MPI\_Comm comm, MPI\_Info info) BIND(C)
   TYPE(MPI\_Comm), INTENT(INOUT) :: comm
   TYPE(MPI\_Info), INTENT(IN) :: info
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- MPI\_COMM\_SET\_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR

MPI\_COMM\_SET\_INFO sets new values for the hints of the communicator associated with comm. MPI\_COMM\_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.

Advice to users. Some info items that an implementation can use when it creates a communicator cannot easily be changed once the communicator has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. (*End of advice to users.*)

MPI\_COMM\_GET\_INFO(comm, info\_used) IN communicator object (handle) comm OUT info\_used new info object (handle) int MPI\_Comm\_get\_info(MPI\_Comm comm, MPI\_Info \*info\_used) MPI\_Comm\_get\_info(comm, info\_used, ierror) BIND(C) TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Info), INTENT(OUT) :: info\_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror 

 $^{24}$ 

#### 1 MPI\_COMM\_GET\_INFO(COMM, INFO\_USED, IERROR) $\mathbf{2}$

INTEGER COMM, INFO\_USED, IERROR

MPI\_COMM\_GET\_INFO returns a new info object containing the hints of the communicator associated with comm. The current setting of all hints actually used by the system related to this communicator is returned in info\_used. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info\_used via MPI\_INFO\_FREE.

Advice to users. The info object returned in info\_used will contain all hints currently active for this communicator. This set of hints may be greater or smaller than the set of hints specified when the communicator was created, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (End of advice to users.)

156.5 16

3

4

5

6

7

8 9

10

11

12

13

14

#### Motivating Examples

176.5.1 Current Practice #1 18 19 Example #1a: 20int main(int argc, char \*argv[]) 21{ 22int me, size; 2324. . . MPI\_Init ( &argc, &argv ); 25MPI\_Comm\_rank (MPI\_COMM\_WORLD, &me); 26MPI\_Comm\_size (MPI\_COMM\_WORLD, &size); 2728(void)printf ("Process %d size %d\n", me, size); 29 30 . . . MPI\_Finalize(); 31return 0; 32 } 33 34

Example #1a is a do-nothing program that initializes itself, and refers to the "all" commu-35 nicator, and prints a message. It terminates itself too. This example does not imply that 36 MPI supports printf-like communication itself.

```
37
      Example #1b (supposing that size is even):
38
```

```
39
         int main(int argc, char *argv[])
40
         {
41
             int me, size;
42
             int SOME_TAG = 0;
43
             . . .
44
            MPI_Init(&argc, &argv);
45
46
            MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
47
             MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
48
```

```
if((me % 2) == 0)
{
    /* send unless highest-numbered process */
    if((me + 1) < size)
        MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
}
else
    MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
...
MPI_Finalize();
return 0;
}</pre>
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

#### 6.5.2 Current Practice #2

```
int main(int argc, char *argv[])
{
  int me, count;
  void *data;
  . . .
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  if(me == 0)
  ſ
      /* get input, create buffer ''data'' */
      . . .
  }
  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
  . . .
  MPI_Finalize();
  return 0;
}
```

This example illustrates the use of a collective communication.

```
6.5.3 (Approximate) Current Practice #3
int main(int argc, char *argv[])
{
    int me, count, count2;
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
```

 $\frac{24}{25}$ 

 $^{31}$ 

```
1
          MPI_Group MPI_GROUP_WORLD, grprem;
\mathbf{2}
          MPI_Comm commslave;
3
          static int ranks[] = {0};
4
          . . .
5
          MPI_Init(&argc, &argv);
6
          MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
\overline{7}
          MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
8
9
          MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem); /* local */
10
          MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);
11
12
          if(me != 0)
13
          {
14
            /* compute on slave */
15
            . . .
16
            MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commslave);
17
18
            MPI_Comm_free(&commslave);
19
          }
20
          /* zero falls through immediately to this reduce, others do later... */
21
          MPI_Reduce(send_buf2, recv_buf2, count2,
22
                      MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
23
24
          MPI_Group_free(&MPI_GROUP_WORLD);
25
          MPI_Group_free(&grprem);
26
          MPI_Finalize();
27
          return 0;
       }
28
29
     This example illustrates how a group consisting of all but the zeroth process of the "all"
30
     group is created, and then how a communicator is formed (commslave) for that new group.
^{31}
     The new communicator is used in a collective call, and all processes execute a collective call
32
     in the MPI_COMM_WORLD context. This example illustrates how the two communicators
33
     (that inherently possess distinct contexts) protect communication. That is, communication
34
     in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.
35
         In summary, "group safety" is achieved via communicators because distinct contexts
36
     within communicators are enforced to be unique on any process.
37
38
39
     6.5.4 Example #4
40
     The following example is meant to illustrate "safety" between point-to-point and collective
41
     communication. MPI guarantees that a single communicator can do safe point-to-point and
42
     collective communication.
43
44
         #define TAG_ARBITRARY 12345
45
         #define SOME_COUNT
                                     50
46
47
         int main(int argc, char *argv[])
48
         {
```

```
1
     int me;
                                                                                        \mathbf{2}
     MPI_Request request[2];
                                                                                        3
     MPI_Status status[2];
     MPI_Group MPI_GROUP_WORLD, subgroup;
                                                                                        4
     int ranks[] = \{2, 4, 6, 8\};
                                                                                        5
                                                                                        6
     MPI_Comm the_comm;
                                                                                        7
     . . .
                                                                                        8
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
                                                                                        9
                                                                                        10
                                                                                        11
     MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
     MPI_Group_rank(subgroup, &me);
                                          /* local */
                                                                                       12
                                                                                        13
                                                                                       14
     MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
                                                                                        15
                                                                                        16
     if(me != MPI_UNDEFINED)
                                                                                        17
     {
                                                                                       18
         MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                                                                                       19
                             the_comm, request);
                                                                                       20
          MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                                                                                       21
                             the_comm, request+1);
          for(i = 0; i < SOME_COUNT; i++)</pre>
                                                                                       22
            MPI_Reduce(..., the_comm);
                                                                                       23
                                                                                        ^{24}
         MPI_Waitall(2, request, status);
                                                                                        25
                                                                                        26
          MPI_Comm_free(&the_comm);
     }
                                                                                       27
                                                                                       28
     MPI_Group_free(&MPI_GROUP_WORLD);
                                                                                       29
                                                                                        30
     MPI_Group_free(&subgroup);
     MPI_Finalize();
                                                                                        ^{31}
                                                                                        32
     return 0;
                                                                                        33
   }
                                                                                       34
                                                                                       35
6.5.5
      Library Example \#1
                                                                                       36
The main program:
                                                                                       37
                                                                                        38
   int main(int argc, char *argv[])
                                                                                        39
   {
                                                                                        40
     int done = 0;
                                                                                        41
     user_lib_t *libh_a, *libh_b;
                                                                                        42
     void *dataset1, *dataset2;
                                                                                        43
     . . .
                                                                                        44
     MPI_Init(&argc, &argv);
                                                                                        45
     . . .
                                                                                        46
     init_user_lib(MPI_COMM_WORLD, &libh_a);
                                                                                        47
     init_user_lib(MPI_COMM_WORLD, &libh_b);
                                                                                        48
```

```
1
           . . .
\mathbf{2}
           user_start_op(libh_a, dataset1);
3
           user_start_op(libh_b, dataset2);
4
           . . .
5
           while(!done)
6
           {
7
              /* work */
8
               . . .
9
              MPI_Reduce(..., MPI_COMM_WORLD);
10
               . . .
11
              /* see if done */
12
               . . .
13
           }
14
           user_end_op(libh_a);
15
           user_end_op(libh_b);
16
17
           uninit_user_lib(libh_a);
18
           uninit_user_lib(libh_b);
19
           MPI_Finalize();
20
           return 0;
21
         }
22
     The user library initialization code:
23
24
         void init_user_lib(MPI_Comm comm, user_lib_t **handle)
25
         {
26
           user_lib_t *save;
27
28
           user_lib_initsave(&save); /* local */
29
           MPI_Comm_dup(comm, &(save -> comm));
30
^{31}
           /* other inits */
32
           . . .
33
34
           *handle = save;
35
         }
36
37
     User start-up code:
38
         void user_start_op(user_lib_t *handle, void *data)
39
         {
40
           MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
^{41}
           MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
42
         }
43
44
     User communication clean-up code:
45
46
         void user_end_op(user_lib_t *handle)
47
         {
48
           MPI_Status status;
```

```
1
     MPI_Wait(& handle -> isend_handle, &status);
                                                                                      \mathbf{2}
     MPI_Wait(& handle -> irecv_handle, &status);
                                                                                      3
   }
                                                                                      4
User object clean-up code:
                                                                                      5
                                                                                      6
   void uninit_user_lib(user_lib_t *handle)
                                                                                      7
   {
                                                                                       8
     MPI_Comm_free(&(handle -> comm));
                                                                                      9
     free(handle);
                                                                                      10
   }
                                                                                      11
                                                                                      12
      Library Example #2
6.5.6
                                                                                      13
                                                                                      14
The main program:
                                                                                      15
   int main(int argc, char *argv[])
                                                                                      16
   Ł
                                                                                      17
     int ma, mb;
                                                                                      18
     MPI_Group MPI_GROUP_WORLD, group_a, group_b;
                                                                                      19
     MPI_Comm comm_a, comm_b;
                                                                                      20
                                                                                      21
     static int list_a[] = {0, 1};
                                                                                      22
#if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
                                                                                      23
     static int list_b[] = {0, 2, 3};
                                                                                      ^{24}
#else/* EXAMPLE_2A */
                                                                                      25
     static int list_b[] = {0, 2};
                                                                                      26
#endif
                                                                                      27
     int size_list_a = sizeof(list_a)/sizeof(int);
                                                                                      28
     int size_list_b = sizeof(list_b)/sizeof(int);
                                                                                      29
                                                                                      30
     . . .
                                                                                      31
     MPI_Init(&argc, &argv);
                                                                                      32
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
                                                                                      33
                                                                                      34
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
                                                                                      35
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);
                                                                                      36
                                                                                      37
     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
                                                                                      38
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
                                                                                      39
                                                                                      40
     if(comm_a != MPI_COMM_NULL)
                                                                                      41
        MPI_Comm_rank(comm_a, &ma);
                                                                                      42
     if(comm_b != MPI_COMM_NULL)
                                                                                      43
        MPI_Comm_rank(comm_b, &mb);
                                                                                      44
                                                                                      45
     if(comm_a != MPI_COMM_NULL)
                                                                                      46
        lib_call(comm_a);
                                                                                      47
```

```
1
           if(comm_b != MPI_COMM_NULL)
\mathbf{2}
           {
3
             lib_call(comm_b);
4
             lib_call(comm_b);
5
           }
6
7
           if(comm_a != MPI_COMM_NULL)
8
             MPI_Comm_free(&comm_a);
9
           if(comm_b != MPI_COMM_NULL)
10
             MPI_Comm_free(&comm_b);
11
           MPI_Group_free(&group_a);
12
           MPI_Group_free(&group_b);
13
           MPI_Group_free(&MPI_GROUP_WORLD);
14
           MPI_Finalize();
15
           return 0;
16
         }
17
     The library:
18
19
         void lib_call(MPI_Comm comm)
20
         ſ
21
           int me, done = 0;
22
           MPI_Status status;
23
           MPI_Comm_rank(comm, &me);
^{24}
           if(me == 0)
25
              while(!done)
26
              {
27
                  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
28
                  . . .
29
              }
30
           else
31
           ſ
32
             /* work */
33
             MPI_Send(..., 0, ARBITRARY_TAG, comm);
34
              . . . .
35
           }
36
     #ifdef EXAMPLE_2C
37
           /* include (resp, exclude) for safety (resp, no safety): */
38
           MPI_Barrier(comm);
39
     #endif
40
         }
41
```

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 back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [57]). 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 back-masking 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.

#### 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 modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called "inter-communication" and the communicator used is called an "inter-communicator," as introduced earlier.

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 except for MPI\_COMM\_IDUP 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

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

### 258 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

1 2 3 4 5	MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the pro- cesses in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to inter- communicators makes the most sense when the groups are disjoint. ( <i>End of advice to</i> <i>users.</i> )
6 7	Here is a summary of the properties of inter-communication and inter-communicators:
8 9 10 11	• The syntax of point-to-point and collective communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
12 13 14	• A target process is addressed by its rank in the remote group, both for sends and for receives.
14 15 16	• Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
17 18	• A communicator will provide either intra- or inter-communication, never both.
19 20 21 22 23	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).
24 25	Advice to implementors. For the purpose of point-to-point communication, commu- nicators can be represented in each process by a tuple consisting of:
26 27	group
28	send_context
29 30	receive_context
31	source
32 33 34 35 36	For inter-communicators, <i>group</i> describes the remote group, and <i>source</i> is the rank of the process in the local group. For intra-communicators, <i>group</i> is the communicator group (remote=local), <i>source</i> is the rank of the process in this group, and <i>send context</i> and <i>receive context</i> are identical. A group can be represented by a rank-to-absolute-address translation table.
37 38 39 40 41	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 inter- communicator $\mathbf{C}_{\mathcal{P}}$ , and a process $\mathbf{Q}$ in group $\mathcal{Q}$ , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$ . Then
42	• $\mathbf{C}_{\mathcal{P}}$ .group describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$ .group describes the group $\mathcal{P}$ .
43 44 45	• $C_{\mathcal{P}}$ .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 $\mathcal{P}$ .
45 46	• $\mathbf{C}_{\mathcal{P}}$ .source is rank of <b>P</b> in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$ .source is rank of <b>Q</b> in $\mathcal{Q}$ .
47 48	

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)
OUT	flag	(logical)

int MPI\_Comm\_test\_inter(MPI\_Comm comm, int \*flag)

```
MPI_Comm_test_inter(comm, flag, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
```

This local routine allows the calling process to determine if a communicator is an intercommunicator or an intra-communicator. It returns true if it is an inter-communicator, otherwise false.

When an inter-communicator is used as an input argument to the communicator accessors described above under intra-communication, the following table describes behavior.

MPI_COMM_SIZE	returns the size of the local group.
MPI_COMM_GROUP	returns the local group.
MPI_COMM_RANK	returns the rank in the local group

Table 6.1: MPI\_COMM\_\* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI\_COMM\_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI\_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI\_CONGRUENT or MPI\_SIMILAR. In particular, it is possible for MPI\_SIMILAR to result because either the local or remote groups were similar but not identical. 

 $^{24}$ 

1 The following accessors provide consistent access to the remote group of an inter- $\mathbf{2}$ communicator. The following are all local operations. 3 4 MPI\_COMM\_REMOTE\_SIZE(comm, size) 56 IN comm inter-communicator (handle)  $\overline{7}$ OUT number of processes in the remote group of comm size 8 (integer) 9 10 int MPI\_Comm\_remote\_size(MPI\_Comm comm, int \*size) 11 12MPI\_Comm\_remote\_size(comm, size, ierror) BIND(C) 13TYPE(MPI\_Comm), INTENT(IN) :: comm 14INTEGER, INTENT(OUT) :: size 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI\_COMM\_REMOTE\_SIZE(COMM, SIZE, IERROR) 17INTEGER COMM, SIZE, IERROR 18 192021MPI\_COMM\_REMOTE\_GROUP(comm, group) 22IN comm inter-communicator (handle) 23OUT remote group corresponding to **comm** (handle) group  $^{24}$ 25int MPI\_Comm\_remote\_group(MPI\_Comm comm, MPI\_Group \*group) 2627MPI\_Comm\_remote\_group(comm, group, ierror) BIND(C) 28TYPE(MPI\_Comm), INTENT(IN) :: comm 29TYPE(MPI\_Group), INTENT(OUT) :: group 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror  $^{31}$ MPI\_COMM\_REMOTE\_GROUP(COMM, GROUP, IERROR) 32 INTEGER COMM, GROUP, IERROR 33 34 35Rationale. Symmetric access to both the local and remote groups of an inter-36 communicator is important, so this function, as well as MPI\_COMM\_REMOTE\_SIZE 37 have been provided. (End of rationale.) 38 39 Inter-communicator Operations 6.6.2 4041This section introduces four blocking inter-communicator operations. 42MPI\_INTERCOMM\_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI\_INTERCOMM\_MERGE creates an intra-communicator by merg-43ing the local and remote groups of an inter-communicator. The functions MPI\_COMM\_DUP  $^{44}$ and MPI\_COMM\_FREE, introduced previously, duplicate and free an inter-communicator, 4546respectively.

<sup>47</sup> Overlap of local and remote groups that are bound into an inter-communicator is <sup>48</sup> 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 from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

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.

In standard MPI implementations (with static process allocation at initialization), the MPI\_COMM\_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that have used spawn or join, it may be necessary to first create an intracommunicator to be used as peer.

The application topology functions described in Chapter 7 do not apply to intercommunicators. Users that require this capability should utilize

MPI\_INTERCOMM\_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topologyoriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

## MPI\_INTERCOMM\_CREATE(local\_comm, local\_leader, peer\_comm, remote\_leader, tag, newintercomm)

IN	local_comm	local intra-communicator (handle)	27		
IN	local_leader	rank of local group leader in local_comm (integer)			
IN	peer_comm	"peer" communicator; significant only at the	29 30		
		local_leader (handle)	31		
IN	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader (integer)	32 33		
IN	tag	tag (integer)	34		
OUT	newintercomm	new inter-communicator (handle)	35 36		
int MPI_I	ntercomm_create(MPI_Comm	<pre>local_comm, int local_leader,</pre>	38		
	MPI_Comm peer_comm, int remote_leader, int tag,				
	MPI_Comm *newintercor	nm)	40		
MPI_Inter	<pre>comm_create(local_comm, l</pre>	local_leader, peer_comm, remote_leader,	41 42		
	tag, newintercomm, ie		43		
	MPI_Comm), INTENT(IN) ::	-	44		
	ER, INTENT(IN) :: local_ MPI_Comm), INTENT(OUT) ::	leader, remote_leader, tag	45		
	ER, OPTIONAL, INTENT(OUT)		46		
10100		101101	47		
			48		

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

1 MPI\_INTERCOMM\_CREATE(LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER,  $\mathbf{2}$ TAG, NEWINTERCOMM, IERROR) 3 INTEGER LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER, TAG, 4 NEWINTERCOMM, IERROR 5This call creates an inter-communicator. It is collective over the union of the local and 6 remote groups. Processes should provide identical local\_comm and local\_leader arguments 7 within each group. Wildcards are not permitted for remote\_leader, local\_leader, and tag. 8 9 10MPI\_INTERCOMM\_MERGE(intercomm, high, newintracomm) 11IN intercomm Inter-Communicator (handle) 12IN 13high (logical) 14OUT newintracomm new intra-communicator (handle) 1516int MPI\_Intercomm\_merge(MPI\_Comm intercomm, int high, 17MPI\_Comm \*newintracomm) 18 19MPI\_Intercomm\_merge(intercomm, high, newintracomm, ierror) BIND(C) 20TYPE(MPI\_Comm), INTENT(IN) :: intercomm 21LOGICAL, INTENT(IN) :: high 22TYPE(MPI\_Comm), INTENT(OUT) :: newintracomm 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI\_INTERCOMM\_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) 25INTEGER INTERCOMM, NEWINTRACOMM, IERROR 26LOGICAL HIGH 2728This function creates an intra-communicator from the union of the two groups that are 29associated with intercomm. All processes should provide the same high value within each 30of the two groups. If processes in one group provided the value high = false and processes  $^{31}$ in the other group provided the value high = true then the union orders the "low" group 32before the "high" group. If all processes provided the same high argument then the order 33 of the union is arbitrary. This call is blocking and collective within the union of the two 34 groups. 35 The error handler on the new intercommunicator in each process is inherited from 36 the communicator that contributes the local group. Note that this can result in different 37 processes in the same communicator having different error handlers. 38 Advice to implementors. The implementation of MPI\_INTERCOMM\_MERGE, 39 MPI\_COMM\_FREE, and MPI\_COMM\_DUP are similar to the implementation of 4041 MPI\_INTERCOMM\_CREATE, except that contexts private to the input inter-com-42municator are used for communication between group leaders rather than contexts inside a bridge communicator. (End of advice to implementors.) 43 44454647 48

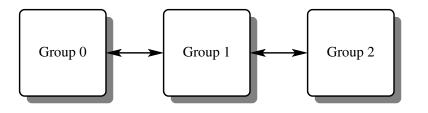


Figure 6.3: Three-group pipeline

#### 6.6.3 Inter-Communication Examples

#### Example 1: Three-Group "Pipeline"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```
int main(int argc, char *argv[])
                                                                                 17
{
                                                                                 18
                            /* intra-communicator of local sub-group */
 MPI_Comm
             myComm;
                                                                                 19
 MPI_Comm
             myFirstComm; /* inter-communicator */
                                                                                 20
 MPI_Comm
             mySecondComm; /* second inter-communicator (group 1 only) */
                                                                                 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

9

10 11

12

13

14

```
1
2
3
                           Group 0
4
                                                                Group 2
                                             Group 1
5
6
7
8
                                   Figure 6.4: Three-group ring
9
10
             MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
11
                                     12, &myFirstComm);
12
           }
13
14
           /* Do work ... */
15
16
           switch(membershipKey) /* free communicators appropriately */
17
           {
18
           case 1:
19
              MPI_Comm_free(&mySecondComm);
20
           case 0:
21
           case 2:
22
              MPI_Comm_free(&myFirstComm);
23
              break;
^{24}
           }
25
26
           MPI_Finalize();
27
           return 0;
28
        }
29
30
     Example 2: Three-Group "Ring"
31
32
     Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate.
     Therefore, each requires two inter-communicators.
33
34
         int main(int argc, char *argv[])
35
         {
36
           MPI_Comm
                                      /* intra-communicator of local sub-group */
                       myComm;
37
           MPI_Comm
                       myFirstComm; /* inter-communicators */
38
           MPI_Comm
                       mySecondComm;
39
           int membershipKey;
40
           int rank;
^{41}
42
           MPI_Init(&argc, &argv);
43
           MPI_Comm_rank(MPI_COMM_WORLD, &rank);
44
           . . .
45
46
           /* User code must generate membershipKey in the range [0, 1, 2] */
47
           membershipKey = rank % 3;
48
```

```
/* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
  /* Build inter-communicators. Tags are hard-coded. */
  if (membershipKey == 0)
  {
                /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                         1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                         2, &mySecondComm);
  }
  else if (membershipKey == 1)
  {
            /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                         1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                         12, &mySecondComm);
  }
  else if (membershipKey == 2)
           /* Group 2 communicates with groups 0 and 1. */
  {
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                         2, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                         12, &mySecondComm);
  }
  /* Do some work ... */
  /* Then free communicators before terminating... */
  MPI_Comm_free(&myFirstComm);
  MPI_Comm_free(&mySecondComm);
  MPI_Comm_free(&myComm);
  MPI_Finalize();
  return 0;
}
```

### 6.7 Caching

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

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

 $45 \\ 46$ 

• 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 attribute-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. For this reason, the Fortran versions of these routines use integers of kind MPI\_ADDRESS\_KIND.

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.)

 $^{24}$ 

#### 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 or MPI\_COMM\_IDUP (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:

- obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoids problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.

A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency "hit" inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

#### 6.7.2 Communicators

Functions for caching on communicators are:

## MPI\_COMM\_CREATE\_KEYVAL(comm\_copy\_attr\_fn, comm\_delete\_attr\_fn, comm\_keyval, extra\_state)

IN	comm_copy_attr_fn	copy callback function for $comm\_keyval$ (function)
IN	comm_delete_attr_fn	delete callback function for $comm\_keyval\xspace$ (function)
OUT	comm_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

MPI\_Comm\_create\_keyval(comm\_copy\_attr\_fn, comm\_delete\_attr\_fn, comm\_keyval, extra\_state, ierror) BIND(C) PROCEDURE(MPI\_Comm\_copy\_attr\_function) :: comm\_copy\_attr\_fn PROCEDURE(MPI\_Comm\_delete\_attr\_function) :: comm\_delete\_attr\_fn INTEGER, INTENT(OUT) :: comm\_keyval INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state INTEGER, OPTIONAL, INTENT(OUT) :: ierror 

```
1
              MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
         \mathbf{2}
                             EXTRA_STATE, IERROR)
         3
                   EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
          4
                   INTEGER COMM_KEYVAL, IERROR
         5
                   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
          6
                   Generates a new attribute key. Keys are locally unique in a process, and opaque to
          7
               user, though they are explicitly stored in integers. Once allocated, the key value can be
          8
               used to associate attributes and access them on any locally defined communicator.
         9
               The C callback functions are:
         10
               typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
         11
                             void *extra_state, void *attribute_val_in,
         12
                             void *attribute_val_out, int *flag);
         13
         14
               and
         15
               typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
         16
                             void *attribute_val, void *extra_state);
         17
               which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
         18
               With the mpi_f08 module, the Fortran callback functions are:
         19
               ABSTRACT INTERFACE
         20
                 SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
         21
                 attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
         22
                     TYPE(MPI_Comm) :: oldcomm
         23
                     INTEGER :: comm_keyval, ierror
         24
                     INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
         25
                     attribute_val_out
         26
                     LOGICAL :: flag
         27
         28
               and
         ^{29}
               ABSTRACT INTERFACE
         30
                 SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
         31
                 attribute_val, extra_state, ierror) BIND(C)
         32
                     TYPE(MPI_Comm) :: comm
         33
                     INTEGER :: comm_keyval, ierror
         34
                     INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
         35
ticketWG. 36
               With the mpi module and the deprecated mpif.h, the Fortran callback functions are:
               SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
         37
                             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
         38
                   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
         39
                   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
         40
                       ATTRIBUTE_VAL_OUT
         41
                   LOGICAL FLAG
         42
         43
               and
         ^{44}
               SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
         45
                             EXTRA_STATE, IERROR)
         46
                   INTEGER COMM, COMM_KEYVAL, IERROR
         47
                   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
         48
```

1 The comm\_copy\_attr\_fn function is invoked when a communicator is duplicated by  $\mathbf{2}$ MPI\_COMM\_DUP or MPI\_COMM\_IDUP. comm\_copy\_attr\_fn should be of type 3 MPI\_Comm\_copy\_attr\_function. The copy callback function is invoked for each key value in 4 oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its  $\mathbf{5}$ 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 6  $\overline{7}$ the value returned in attribute\_val\_out. The function returns MPI\_SUCCESS on success and an error code on failure (in which case MPI\_COMM\_DUP or MPI\_COMM\_IDUP will fail). 8

The argument comm\_copy\_attr\_fn may be specified as MPI\_COMM\_NULL\_COPY\_FN or MPI\_COMM\_DUP\_FN from either C or Fortran. MPI\_COMM\_NULL\_COPY\_FN is a function that does nothing other than returning flag = 0 or .FALSE. (depending on whether 12the keyval was created with a C or Fortran binding 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 MPI\_NULL\_COPY\_FN and MPI\_DUP\_FN, whose use is deprecated.

Even though both formal arguments attribute\_val\_in and Advice to users. 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.

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.)

A C interface should be assumed for copy and delete Advice to implementors. 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 41 42MPI\_COMM\_NULL\_DELETE\_FN from either C or Fortran. MPI\_COMM\_NULL\_DELETE\_FN is a function that does nothing, other than returning 43 MPI\_SUCCESS. MPI\_COMM\_NULL\_DELETE\_FN replaces MPI\_NULL\_DELETE\_FN, whose 44use is deprecated. 4546

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.

9

10 11

13

1415

16

17

18

19

20

21

2223

 $^{24}$ 

25

26

27

28 29

30

 $^{31}$ 

32 33

34

35

36

37

38

39

40

```
1
                    The special key value MPI_KEYVAL_INVALID is never returned by
          \mathbf{2}
                MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key
          3
               values.
          4
                     Advice to implementors.
                                                  The predefined Fortran functions
          5
                     MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
          6
ticketWG.
                     MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and the deprecated
                     mpif.h) and the mpi_f08 module with the same name, but with different interfaces.
          8
                     Each function can coexist twice with the same name in the same MPI library, one
          9
                     routine as an implicit interface outside of the mpi module, i.e., declared as EXTERNAL,
          10
                     and the other routine within mpi_f08 declared with CONTAINS. These routines have
          11
                     different link names, which are also different to the link names used for the routines
          12
                     used in C. (End of advice to implementors.)
          13
                                        Callbacks, including the predefined Fortran functions
          14
                     Advice to users.
                     MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
          15
                     MPI_COMM_NULL_DELETE_FN should not be passed from one application routine
          16
                     that uses the mpi_f08 module to another application routine that uses the mpi module
          17
ticketWG. 18
                     or the deprecated mpif.h, and vice versa; see also the advice to users on page 656.
          19
                     (End of advice to users.)
          20
          21
          22
               MPI_COMM_FREE_KEYVAL(comm_keyval)
          23
                 INOUT
                           comm_keyval
                                                        key value (integer)
          ^{24}
          25
                int MPI_Comm_free_keyval(int *comm_keyval)
          26
          27
               MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)
          28
                    INTEGER, INTENT(INOUT) :: comm_keyval
          29
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
          30
               MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
          ^{31}
                    INTEGER COMM_KEYVAL, IERROR
          32
          33
                    Frees an extant attribute key. This function sets the value of keyval to
          34
                MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use.
          35
                because the actual free does not transpire until after all references (in other communicators
          36
                on the process) to the key have been freed. These references need to be explicitly freed by the
          37
                program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,
          38
                or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
          39
                communicator.
          40
          41
               MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)
          42
          43
                 INOUT
                                                        communicator from which attribute will be attached
                           comm
          44
                                                        (handle)
          45
                 IN
                           comm_keyval
                                                        key value (integer)
          46
                 IN
                           attribute val
                                                        attribute value
          47
          48
```

#### 6.7. CACHING

int MPI_0	Comm_set_attr(MPI_Comm co	mm, int comm_keyval, void *attribute_val)	1		
MPI Comm	set attr(comm. comm kevv	al, attribute_val, ierror) BIND(C)	2		
	(MPI_Comm), INTENT(IN) ::		$\frac{3}{4}$		
	GER, INTENT(IN) :: comm_		4 5		
INTE	GER(KIND=MPI_ADDRESS_KIND	), INTENT(IN) :: attribute_val	6		
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	7		
MPI COMM	SET ATTR(COMM. COMM KEYV	AL, ATTRIBUTE_VAL, IERROR)	8		
	GER COMM, COMM_KEYVAL, IE		9		
	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL				
$T_{bin}$	function stores the stimulated	ttuibute velue attuibute und fen subgesuent netnievel	11		
	-	attribute value attribute_val for subsequent retrieval lue is already present, then the outcome is as if	12		
		called to delete the previous value (and the callback	13		
		uted), and a new value was next stored. The call	14 15		
		ie keyval; in particular MPI_KEYVAL_INVALID is an	15		
erroneous	key value. The call will fail if	the comm_delete_attr_fn function returned an error	17		
code other	$ than MPI_SUCCESS. $		18		
			19		
MPL COM	IM_GET_ATTR(comm, comm_	keyval attribute val flag)	20		
		,	21		
IN	comm	communicator to which the attribute is attached (han-	22		
		dle)	23		
IN	comm_keyval	key value (integer)	24		
OUT	attribute_val	attribute value, unless $flag = false$	25 26		
OUT	flag	false if no attribute is associated with the key (logical)	20		
			28		
int MPI_(	Comm_get_attr(MPI_Comm co	mm, int comm_keyval, void *attribute_val,	29		
	int *flag)		30		
MPT Comm	get attr(comm comm keyy	al, attribute_val, flag, ierror) BIND(C)	31		
	(MPI_Comm), INTENT(IN) ::		32		
	GER, INTENT(IN) :: comm_		33		
INTE	GER(KIND=MPI_ADDRESS_KIND	), INTENT(OUT) :: attribute_val	34		
LOGI	CAL, INTENT(OUT) :: flag		35		
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	$\frac{36}{37}$		
MPT COMM	GET ATTR(COMM, COMM KEYV	AL, ATTRIBUTE_VAL, FLAG, IERROR)	38		
	GER COMM, COMM_KEYVAL, IE		39		
	GER(KIND=MPI_ADDRESS_KIND		40		
LOGI	CAL FLAG		41		
Retrie	eves attribute value by key '	The call is erroneous if there is no key with value	42		
		correct if the key value exists, but no attribute is	43		

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns flag = false. In particular MPI\_KEYVAL\_INVALID is an erroneous key value.

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 48

271

44

45

location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void\*, then the actual attribute\_val parameter to MPI\_Comm\_set\_attr will be of type void\* and the actual attribute\_val parameter to MPI\_Comm\_get\_attr will be of type void\*\*. (*End of advice to users.*)

*Rationale.* The use of a formal parameter attribute\_val of type void\* (rather than void\*\*) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void\*. (*End of rationale.*)

9 10 11

1

 $\mathbf{2}$ 

3

4

5

6

7

8

MPI\_COMM\_DELETE\_ATTR(comm, comm\_keyval)

12INOUT communicator from which the attribute is deleted (hancomm 13 dle) 14IN comm\_keyval key value (integer) 1516int MPI\_Comm\_delete\_attr(MPI\_Comm comm, int comm\_keyval) 1718 MPI\_Comm\_delete\_attr(comm, comm\_keyval, ierror) BIND(C) 19TYPE(MPI\_Comm), INTENT(IN) :: comm 20INTEGER, INTENT(IN) :: comm\_keyval 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22MPI\_COMM\_DELETE\_ATTR(COMM, COMM\_KEYVAL, IERROR) 23INTEGER COMM, COMM\_KEYVAL, IERROR  $^{24}$ 25Delete attribute from cache by key. This function invokes the attribute delete function 26comm\_delete\_attr\_fn specified when the keyval was created. The call will fail if the 27comm\_delete\_attr\_fn function returns an error code other than MPI\_SUCCESS. 28Whenever a communicator is replicated using the function MPI\_COMM\_DUP or 29MPI\_COMM\_IDUP, all call-back copy functions for attributes that are currently set are 30 invoked (in arbitrary order). Whenever a communicator is deleted using the function  $^{31}$ MPI\_COMM\_FREE all callback delete functions for attributes that are currently set are 32 invoked. 33 346.7.3 Windows 35 36 The functions for caching on windows are: 37 38 MPI\_WIN\_CREATE\_KEYVAL(win\_copy\_attr\_fn, win\_delete\_attr\_fn, win\_keyval, extra\_state) 39 4041 IN win\_copy\_attr\_fn copy callback function for win\_keyval (function) 42IN win\_delete\_attr\_fn delete callback function for win\_keyval (function) 43OUT win\_keyval key value for future access (integer) 44IN extra state for callback functions extra\_state 454647int MPI\_Win\_create\_keyval(MPI\_Win\_copy\_attr\_function \*win\_copy\_attr\_fn, 48 MPI\_Win\_delete\_attr\_function \*win\_delete\_attr\_fn,

<pre>int *win_keyval, void *extra_state)</pre>	1
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,	2 3
extra_state, ierror) BIND(C)	4
PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn	5
PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn	6
INTEGER, INTENT(OUT) :: win_keyval	7
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	10
EXTRA_STATE, IERROR)	11
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	12
INTEGER WIN_KEYVAL, IERROR	13
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	14
The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or	15
MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function	16
that does nothing other than returning $flag = 0$ and MPI_SUCCESS. MPI_WIN_DUP_FN is	17
a simple-minded copy function that sets $flag = 1$ , returns the value of attribute_val_in in	18
attribute_val_out, and returns MPI_SUCCESS.	19
The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN	20 21
from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,	21
other than returning MPI_SUCCESS.	22
The C callback functions are:	24
<pre>typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,</pre>	25
<pre>void *extra_state, void *attribute_val_in,</pre>	26
<pre>void *attribute_val_out, int *flag);</pre>	27
and	28
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,	29
void *attribute_val, void *extra_state);	30
	31
With the mpi_f08 module, the Fortran callback functions are:	32
ABSTRACT INTERFACE	33
SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state, attribute_val_in, attribute_val_out, flag, ierror) BIND(C)	34
TYPE(MPI_Win) :: oldwin	35
INTEGER :: win_keyval, ierror	36 37
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,	38
attribute_val_out	39
LOGICAL :: flag	40
and	41
and Abstract interface	42
SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,	43
extra_state, ierror) BIND(C)	44
TYPE(MPI_Win) :: win	45
INTEGER :: win_keyval, ierror	46
INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state	47
	48

```
ticketWG. <sup>1</sup>
               With the mpi module and the deprecated mpif.h, the Fortran callback functions are:
          \mathbf{2}
               SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
          3
                              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
          4
                   INTEGER OLDWIN, WIN_KEYVAL, IERROR
          \mathbf{5}
                   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
          6
                        ATTRIBUTE_VAL_OUT
          7
                   LOGICAL FLAG
          8
               and
          9
               SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
          10
                              EXTRA_STATE, IERROR)
          11
                   INTEGER WIN, WIN_KEYVAL, IERROR
         12
                   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
         13
         14
                   If an attribute copy function or attribute delete function returns other than
          15
               MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
          16
               erroneous.
          17
          18
               MPI_WIN_FREE_KEYVAL(win_keyval)
         19
         20
                 INOUT
                          win_keyval
                                                      key value (integer)
         21
         22
               int MPI_Win_free_keyval(int *win_keyval)
         23
               MPI_Win_free_keyval(win_keyval, ierror) BIND(C)
         ^{24}
                    INTEGER, INTENT(INOUT) :: win_keyval
         25
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
         26
         27
               MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
         28
                   INTEGER WIN_KEYVAL, IERROR
         29
         30
         ^{31}
               MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
         32
         33
                 INOUT
                                                      window to which attribute will be attached (handle)
                          win
         34
                 IN
                          win_keyval
                                                      key value (integer)
         35
                 IN
                          attribute_val
                                                      attribute value
         36
         37
               int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
         38
         39
               MPI_Win_set_attr(win, win_keyval, attribute_val, ierror) BIND(C)
         40
                   TYPE(MPI_Win), INTENT(IN) :: win
         41
                   INTEGER, INTENT(IN) :: win_keyval
         42
                   INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
         43
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
         44
               MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
         45
                   INTEGER WIN, WIN_KEYVAL, IERROR
          46
         47
                   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
          48
```

MPI\_WIN\_GET\_ATTR(win, win\_keyval, attribute\_val, flag) IN win window to which the attribute is attached (handle) win\_keyval IN key value (integer) OUT attribute\_val attribute value, unless flag = falseOUT flag false if no attribute is associated with the key (logical) int MPI\_Win\_get\_attr(MPI\_Win win, int win\_keyval, void \*attribute\_val, int \*flag) 11 MPI\_Win\_get\_attr(win, win\_keyval, attribute\_val, flag, ierror) BIND(C) TYPE(MPI\_Win), INTENT(IN) :: win 1213 INTEGER, INTENT(IN) :: win\_keyval 14INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_WIN\_GET\_ATTR(WIN, WIN\_KEYVAL, ATTRIBUTE\_VAL, FLAG, IERROR) 18 INTEGER WIN, WIN\_KEYVAL, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL 20LOGICAL FLAG 2122 23MPI\_WIN\_DELETE\_ATTR(win, win\_keyval)  $^{24}$ 25INOUT window from which the attribute is deleted (handle) win 26IN win\_keyval key value (integer) 2728int MPI\_Win\_delete\_attr(MPI\_Win win, int win\_keyval) 29 30 MPI\_Win\_delete\_attr(win, win\_keyval, ierror) BIND(C) TYPE(MPI\_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: win\_keyval INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34MPI\_WIN\_DELETE\_ATTR(WIN, WIN\_KEYVAL, IERROR) 35 INTEGER WIN, WIN\_KEYVAL, IERROR 36 37

#### 6.7.4 Datatypes

The new functions for caching on datatypes are:

1  $\mathbf{2}$ 

3

4

56

7 8

9

10

15

16

17

19

31

32

33

38

39

276 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
     MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
\mathbf{2}
                    extra_state)
3
       IN
                type_copy_attr_fn
                                           copy callback function for type_keyval (function)
4
       IN
                type_delete_attr_fn
                                           delete callback function for type_keyval (function)
5
6
       OUT
                type_keyval
                                           key value for future access (integer)
7
       IN
                extra_state
                                           extra state for callback functions
8
9
     int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
10
                    MPI_Type_delete_attr_function *type_delete_attr_fn,
11
                    int *type_keyval, void *extra_state)
12
13
     MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
14
                    extra_state, ierror) BIND(C)
15
         PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
16
         PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
17
         INTEGER, INTENT(OUT) :: type_keyval
18
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
21
                    EXTRA_STATE, IERROR)
22
         EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
23
         INTEGER TYPE_KEYVAL, IERROR
^{24}
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
25
26
         The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or
27
     MPI_TYPE_DUP_FN from either C or Fortran. MPI_TYPE_NULL_COPY_FN is a function
28
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN
^{29}
     is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
30
     attribute_val_out, and returns MPI_SUCCESS.
^{31}
         The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN
32
     from either C or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing,
33
     other than returning MPI_SUCCESS.
34
     The C callback functions are:
35
     typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
36
                    int type_keyval, void *extra_state, void *attribute_val_in,
37
                    void *attribute_val_out, int *flag);
38
     and
39
     typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
40
                    int type_keyval, void *attribute_val, void *extra_state);
41
42
     With the mpi_f08 module, the Fortran callback functions are:
43
     ABSTRACT INTERFACE
44
       SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
45
       attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
46
            TYPE(MPI_Datatype) :: oldtype
47
            INTEGER :: type_keyval, ierror
48
```

INTEGER(KIND=MP	'I_ADDRESS_KIND) :: extr	ra_state, attribute_val_in,	1			
attribute_val_o			2			
LOGICAL :: fla	·g		3			
and			4			
ABSTRACT INTERFACE			5			
	e_delete_attr_function(da	atatype type keyval	6			
	ra_state, ierror) BIND(C)		7			
	rpe) :: datatype		8			
•	pe_keyval, ierror		9			
	PI_ADDRESS_KIND) :: attr	ribute_val, extra_state	10			
		·	11			
-	/	e Fortran callback functions are:	$^{12}$ ticketWG.			
		TYPE_KEYVAL, EXTRA_STATE,	13			
	TE_VAL_IN, ATTRIBUTE_VAL	_OUT, FLAG, IERROR)	14			
	TYPE_KEYVAL, IERROR		16			
	ADDRESS_KIND) EXTRA_STAT	lΕ,	17			
	_IN, ATTRIBUTE_VAL_OUT		18			
LOGICAL FLAG			19			
and			20			
SUBROUTINE TYPE_DELET	E_ATTR_FUNCTION(DATATYPE	E, TYPE_KEYVAL, ATTRIBUTE_VAL,	21			
EXTRA_ST	TATE, IERROR)		22			
INTEGER DATATYPE,	TYPE_KEYVAL, IERROR		23			
INTEGER(KIND=MPI_	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE					
If an attribute copy	If an attribute copy function or attribute delete function returns other than					
		(for example, MPI_TYPE_FREE),	26			
is erroneous.	an that caused it to be more		27			
is cironeous.			28			
			29			
MPI_TYPE_FREE_KEYV	AL(type_keyval)		30			
INOUT type_keyval	key value (in	teger)	31			
	ney varae (iii	(10,000)	32			
int MPI_Type_free_key	wal(int three karwal)		33			
int MF1_Type_Tree_key	Var(int *type_keyvar)		34			
MPI_Type_free_keyval(	type_keyval, ierror) BIN	ID(C)	35			
INTEGER, INTENT(I	NOUT) :: type_keyval		36			
INTEGER, OPTIONAL	., INTENT(OUT) :: ierron		37			
MPI_TYPE_FREE_KEYVAL(	TYPE KEYVAL TEBROR)		38			
INTEGER TYPE_KEYV			39			
			40			
			41			
			42			
			43			
			44			
			45			
			46			

```
278 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING
```

1 MPI\_TYPE\_SET\_ATTR(datatype, type\_keyval, attribute\_val)  $\mathbf{2}$ INOUT datatype datatype to which attribute will be attached (handle) 3 type\_keyval IN key value (integer) 4 5attribute\_val IN attribute value 6  $\overline{7}$ int MPI\_Type\_set\_attr(MPI\_Datatype datatype, int type\_keyval, 8 void \*attribute\_val) 9 MPI\_Type\_set\_attr(datatype, type\_keyval, attribute\_val, ierror) BIND(C) 10 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 11 INTEGER, INTENT(IN) :: type\_keyval 12INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: attribute\_val 13INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_TYPE\_SET\_ATTR(DATATYPE, TYPE\_KEYVAL, ATTRIBUTE\_VAL, IERROR) 16INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 17INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL 18 1920MPI\_TYPE\_GET\_ATTR(datatype, type\_keyval, attribute\_val, flag) 21IN datatype datatype to which the attribute is attached (handle) 22 23IN type\_keyval key value (integer)  $^{24}$ attribute\_val OUT attribute value, unless flag = false2526OUT flag false if no attribute is associated with the key (logical) 2728int MPI\_Type\_get\_attr(MPI\_Datatype datatype, int type\_keyval, void 29\*attribute\_val, int \*flag) 30 MPI\_Type\_get\_attr(datatype, type\_keyval, attribute\_val, flag, ierror)  $^{31}$ BIND(C) 32 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 33 INTEGER, INTENT(IN) :: type\_keyval 34 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val 35 LOGICAL, INTENT(OUT) :: flag 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 38 MPI\_TYPE\_GET\_ATTR(DATATYPE, TYPE\_KEYVAL, ATTRIBUTE\_VAL, FLAG, IERROR) 39 INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 40INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL 41 LOGICAL FLAG 4243 44454647 48

MPI_TYPE	E_DELETE_A	TTR(datatype, ty	pe_keyval)	1
INOUT	datatype		datatype from which the attribute is deleted (handle)	2
IN	type_keyval		1 1 ( <b>:</b> +)	3 4
	51 5			5
int MPI_T	ype_delete_	_attr(MPI_Datat	ype datatype, int type_keyval)	6
MPI_Type_	delete_attr	(datatype, typ	e kevval, ierror) BIND(C)	7
		pe), INTENT(IN)	:: datatype	8 9
		(IN) :: type_k		10
INTEG	ER, UPTIONA	AL, INTENT(OUT)	:: lerror	11
				12
INTEG	ER DATATYPE	E, TYPE_KEYVAL,		13 14
				15
6.7.5 Err	or Class for I	nvalid Keyval		16
Key values	for attribute	es are system-allo	cated, by	17
		,	YVAL. Only such values can be passed to the func-	18 19
	-		ents. In order to signal that an erroneous key value	20
-				21
	e e	,	M,WIN}_DELETE_ATTR,	22
			MPI_{IYPE,COMM,WIN}_GET_ATTR,	23 24
		-	AL, MIPI_COMINI_DUP, MIPI_COMINI_IDUP,	25
		,		26
,	0	10		27
6.7.6 Att	ributes Exan	nple		28 29
Advi	ce to users.	This example		30
opera	ation that use	es caching to be r	more efficient after the first call. (End of advice to	31
users	s.)			32
				33 34
/* key	for this m	nodule's stuff:	*/	35
static	: int gop_ke	ey = MPI_KEYVAL	_INVALID;	36
tunede	f struct			37
typede {	ST SULUCU			38 39
int	ref_count;	; /* r	coforonco count */	40
		, whatever els	e we want */	41
} gop_	stuff_type;			42
void E	Efficient Co	ollective Op (M	IPT Comm comm )	43
{				$\frac{44}{45}$
		*gop_stuff;		46
	Group	group; foundflag;		47
int		roundriag,		48

```
1
2
          MPI_Comm_group(comm, &group);
3
4
          if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
5
          ſ
6
            if ( ! MPI_Comm_create_keyval( gop_stuff_copier,
7
                                       gop_stuff_destructor,
8
                                       &gop_key, (void *)0));
9
            /* get the key while assigning its copy and delete callback
10
               behavior. */
11
12
            MPI_Abort (comm, 99);
          }
13
14
15
          MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
16
          if (foundflag)
17
          { /* This module has executed in this group before.
18
               We will use the cached information */
19
          }
20
          else
21
          { /* This is a group that we have not yet cached anything in.
22
               We will now do so.
23
            */
24
25
            /* First, allocate storage for the stuff we want,
26
               and initialize the reference count */
27
28
            gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
29
            if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
30
31
            gop_stuff -> ref_count = 1;
32
33
            /* Second, fill in *gop_stuff with whatever we want.
34
               This part isn't shown here */
35
36
            /* Third, store gop_stuff as the attribute value */
37
            MPI_Comm_set_attr (comm, gop_key, gop_stuff);
38
          }
39
          /* Then, in any case, use contents of *gop_stuff
40
             to do the global op ... */
41
        }
42
43
        /* The following routine is called by MPI when a group is freed */
44
45
        int gop_stuff_destructor (MPI_Comm comm, int keyval, void *gop_stuffP,
46
                               void *extra)
47
        {
48
          gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
```

```
if (keyval != gop_key) { /* abort -- programming error */ }
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff -> ref_count -= 1;
  /* If no references remain, then free the storage */
  if (gop_stuff -> ref_count == 0) {
    free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}
/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier (MPI_Comm comm, int keyval, void *extra,
 void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in -> ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

#### 6.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

MPI_COMM_SET_NAME (comm, comm_name)						
INOUT comm communicator whose identifier is to be set (handle						
IN	comm_name	the character string which is remembered as the name	38 39			
		(string)	40			
<pre>int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)</pre>						
MPI_Comm_set_name(comm, comm_name, ierror) BIND(C)						
	TYPE(MPI_Comm), INTENT(IN) :: comm					
	CTER(LEN=*), INTENT(IN)	—	46			
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	47			
MPI_COMM_	SET_NAME(COMM, COMM_NAME	, IERROR)	48			

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

1 2		ER COMM, CTER*(*)	IERROR COMM_NAME					
3 4 5 6 7 8 9 10 11	The charac MPI librar stack). Lea MPI_C name of the	eter string y (so it can ading space COMM_SE e communi e is no requ	which is passed to n be freed by the cases in name are sign T_NAME is a local icator as seen in the irrement that the s	ser to associate a name string with a communicator. MPI_COMM_SET_NAME will be saved inside the aller immediately after the call, or allocated on the ificant but trailing ones are not. (non-collective) operation, which only affects the process which made the MPI_COMM_SET_NAME ame (or any) name be assigned to a communicator				
12 13 14 15	is ser	-		MM_SET_NAME is provided to help debug code, it to a communicator in all of the processes where it <i>f advice to users.</i> )				
16 17 18 19 20	MPI_MAX_ null termin	OBJECT_N nator. Att	AME in Fortran an empts to put name	a be stored is limited to the value of d MPI_MAX_OBJECT_NAME-1 in C to allow for the es longer than this will result in truncation of the nave a value of at least 64.				
21 22 23 24 25	Advice to users. Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed. ( <i>End of advice to users.</i> )							
26 27 28 29 30 31 32 33	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. ( <i>End of advice to implementors.</i> )							
34 35	MPI_COM	M_GET_N	AME (comm, comn	n_name, resultlen)				
36	IN	comm		communicator whose name is to be returned (handle)				
37 38 39	OUT	comm_na	ame	the name previously stored on the communicator, or an empty string if no such name exists (string)				
40	OUT	resultlen		length of returned name (integer)				
41 42	int MPI_C	omm_get_r	name(MPI_Comm co	mm, char *comm_name, int *resultlen)				
43 44 45 46 47 48	<pre>MPI_Comm_get_name(comm, comm_name, resultlen, ierror) BIND(C)     TYPE(MPI_Comm), INTENT(IN) :: comm     CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name     INTEGER, INTENT(OUT) :: resultlen     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>							

# MPI\_COMM\_GET\_NAME(COMM, COMM\_NAME, RESULTLEN, IERROR) INTEGER COMM, RESULTLEN, IERROR CHARACTER\*(\*) COMM\_NAME

MPI\_COMM\_GET\_NAME returns the last name which has previously been associated with the given communicator. The name may be set and retrieved from any language. The same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI\_MAX\_OBJECT\_NAME characters. MPI\_COMM\_GET\_NAME returns a copy of the set name in name.

In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI\_MAX\_OBJECT\_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI\_MAX\_OBJECT\_NAME.

If the user has not associated a name with a communicator, or an error occurs, MPI\_COMM\_GET\_NAME will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI\_COMM\_WORLD, MPI\_COMM\_SELF, and the communicator returned by MPI\_COMM\_GET\_PARENT (if not MPI\_COMM\_NULL) will have the default of 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 setting a name on the same communicator; doing this removes the old name and assigns the new one.

*Rationale.* We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.
- To make the attribute key useful additional code to call strdup is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

#### (End of rationale.)

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 <sup>42</sup> an MPI program, and will (necessarily) increase the store requirement of the program, <sup>43</sup> since the names must be saved. Therefore there is no requirement that users use these <sup>44</sup> functions to associate names with communicators. However debugging and profiling <sup>45</sup> MPI applications may be made easier if names are associated with communicators, <sup>46</sup> since the debugger or profiler should then be able to present information in a less <sup>47</sup> cryptic manner. (*End of advice to users.*) <sup>48</sup>

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

1 The following functions are used for setting and getting names of datatypes. The  $\mathbf{2}$ constant MPI\_MAX\_OBJECT\_NAME also applies to these names. 3 4 MPI\_TYPE\_SET\_NAME (datatype, type\_name) 56 INOUT datatype datatype whose identifier is to be set (handle) 7 IN the character string which is remembered as the name type\_name 8 (string) 9 10 int MPI\_Type\_set\_name(MPI\_Datatype datatype, const char \*type\_name) 11 12MPI\_Type\_set\_name(datatype, type\_name, ierror) BIND(C) 13TYPE(MPI\_Datatype), INTENT(IN) :: datatype 14CHARACTER(LEN=\*), INTENT(IN) :: type\_name 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI\_TYPE\_SET\_NAME(DATATYPE, TYPE\_NAME, IERROR) 17INTEGER DATATYPE, IERROR 18 CHARACTER\*(\*) TYPE\_NAME 19 2021MPI\_TYPE\_GET\_NAME (datatype, type\_name, resultlen) 2223IN datatype datatype whose name is to be returned (handle) 24OUT the name previously stored on the datatype, or a empty type\_name 25string if no such name exists (string) 26OUT resultlen length of returned name (integer) 272829int MPI\_Type\_get\_name(MPI\_Datatype datatype, char \*type\_name, int 30 \*resultlen)  $^{31}$ 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 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_TYPE\_GET\_NAME(DATATYPE, TYPE\_NAME, RESULTLEN, IERROR) 38 INTEGER DATATYPE, RESULTLEN, IERROR 39 CHARACTER\*(\*) TYPE\_NAME 40 Named predefined datatypes have the default names of the datatype name. For exam-41 ple, MPI\_WCHAR has the default name of MPI\_WCHAR. 42

The following functions are used for setting and getting names of windows. The constant MPI\_MAX\_OBJECT\_NAME also applies to these names.

- $45 \\ 46$
- 47
- 48

			1			
	MPI_WIN_SET_NAME (win, win_name)					
INOUT	win	window whose identifier is to be set (handle)	3			
IN	win_name	the character string which is remembered as the name	4			
		(string)	5			
			6			
int MPI_	Win_set_name(MPI_Win win,	const char *win_name)	7			
MPT Win	set_name(win, win_name, i	error) BIND(C)	8			
	(MPI_Win), INTENT(IN) ::		9			
	ACTER(LEN=*), INTENT(IN)		10			
	GER, OPTIONAL, INTENT(OUT		11			
			12			
	SET_NAME(WIN, WIN_NAME, I	EKKUK)	13 14			
	GER WIN, IERROR ACTER*(*) WIN_NAME		15			
OIIAN.	ACTER*(*) WIN_NAME		16			
			17			
	_GET_NAME (win, win_name,	resultion)	18			
	Ϋ́Υ.	,	19			
IN	win	window whose name is to be returned (handle)	20			
OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)	21 22			
<u> </u>			22			
OUT	resultlen	length of returned name (integer)	24			
			25			
int MPI_	Win_get_name(MPI_Win win,	char *win_name, int *resultlen)	26			
MPI_Win_	get_name(win, win_name, r	esultlen, ierror) BIND(C)	27			
TYPE	(MPI_Win), INTENT(IN) ::	win	28			
CHAR	ACTER(LEN=MPI_MAX_OBJECT_1	NAME), INTENT(OUT) :: win_name	29			
INTE	GER, INTENT(OUT) :: resu	ltlen	30			
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	31			
MPT WTN	GET_NAME(WIN, WIN_NAME, R	ESULTLEN, TERROR)	32			
	GER WIN, RESULTLEN, IERRO		33			
	ACTER*(*) WIN_NAME		34			
			35 36			
			37			
6.9 Fo	rmalizing the Loosely Sy	nchronous Model	38			

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

39

40

 $\frac{41}{42}$ 

communicator, and under the above guarantee knows that no other communications 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.

<sup>5</sup> This form of safety is analogous to other common computer-science usages, such as <sup>6</sup> passing a descriptor of an array to a library routine. The library routine has every right to <sup>7</sup> expect such a descriptor to be valid and modifiable.

8 9

10

#### 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.

20

#### 21 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 singlethreaded, 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 be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

31

## <sup>32</sup><sub>33</sub> 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.

- 47
- 48

- 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).

#### The General 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, communicator creation is properly coordinated. 

1					
2					
3					
4					
5					
6					
7					
8 9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20 21					
21					
23					
24					
25					
26					
27					
28					
29					
30					
31					
32 33					
33 34					
35					
36					
37					
38					
39					
40					
41					
42					
43					
$44 \\ 45$					
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.

 $^{24}$ 

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 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].

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.

16Specifying the virtual topology in terms of a graph is sufficient for all applications. 17However, in many applications the graph structure is regular, and the detailed set-up of the 18 graph would be inconvenient for the user and might be less efficient at run time. A large frac-19tion of all parallel applications use process topologies like rings, two- or higher-dimensional 20grids, or tori. These structures are completely defined by the number of dimensions and 21the numbers of processes in each coordinate direction. Also, the mapping of grids and tori 22is generally an easier problem than that of general graphs. Thus, it is desirable to address 23these cases explicitly.

<sup>24</sup> Process coordinates in a Cartesian structure begin their numbering at 0. Row-major <sup>25</sup> numbering is always used for the processes in a Cartesian structure. This means that, for <sup>26</sup> example, the relation between group rank and coordinates for four processes in a  $(2 \times 2)$ <sup>27</sup> grid is as follows.

coord (0,0): coord (0,1): coord (1,0):

coord (1,1):

31 32 33

34

35 36

37

38

39

29

30

#### 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.

40 41

42

7.4

#### 7.4 Overview of the Functions

rank 0

rank 1

rank 2

rank 3

<sup>43</sup> MPI supports three topology types: Cartesian, graph, and distributed graph. The function MPI\_CART\_CREATE is used to create Cartesian topologies, the function

<sup>45</sup> MPI\_GRAPH\_CREATE is used to create graph topologies, and the functions

<sup>46</sup> MPI\_DIST\_GRAPH\_CREATE\_ADJACENT and MPI\_DIST\_GRAPH\_CREATE are used to cre-

 $\frac{4}{48}$  at distributed graph topologies. These topology creation functions are collective. As with

1

2

3

 $\frac{4}{5}$ 

6 7

8

9

10

11

12

13

14

other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The 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 MPI\_GRAPH\_CREATE and MPI\_CART\_CREATE, all input arguments must have identical values on all processes of the group of comm\_old. When calling MPI\_GRAPH\_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI\_DIST\_GRAPH\_CREATE\_ADJACENT or MPI\_DIST\_GRAPH\_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm\_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI\_COMM\_CREATE, no cached information propagates 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.*)

MPI defines functions to query a communicator for topology information. The function MPI\_TOPO\_TEST is used to query for the type of topology associated with a communicator. Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI\_GRAPHDIMS\_GET and MPI\_GRAPH\_GET return the values that were specified in the call to MPI\_GRAPH\_CREATE. Additionally, the functions MPI\_GRAPH\_NEIGHBORS\_COUNT 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\_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.

The two additional functions, MPI\_GRAPH\_MAP and MPI\_CART\_MAP, are, in general, 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,

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

44

45

46

47

```
1
     MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the
\mathbf{2}
     nearest neighbors on the topology associated with the communicator. The nonblocking
3
     variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,
4
     MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and
5
     MPI_INEIGHBOR_ALLTOALLW.
6
\overline{7}
            Topology Constructors
     7.5
8
9
     7.5.1 Cartesian Constructor
10
11
12
     MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
13
14
       IN
                  comm_old
                                             input communicator (handle)
15
       IN
                  ndims
                                             number of dimensions of Cartesian grid (integer)
16
       IN
                  dims
                                             integer array of size ndims specifying the number of
17
                                             processes in each dimension
18
19
       IN
                  periods
                                             logical array of size ndims specifying whether the grid
                                             is periodic (true) or not (false) in each dimension
20
21
       IN
                  reorder
                                             ranking may be reordered (true) or not (false) (logical)
22
       OUT
                                             communicator with new Cartesian topology (handle)
                 comm_cart
23
^{24}
     int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const
25
                     int periods[], int reorder, MPI_Comm *comm_cart)
26
27
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
28
                    BIND(C)
29
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
30
          INTEGER, INTENT(IN) :: ndims, dims(ndims)
31
          LOGICAL, INTENT(IN) ::
                                     periods(ndims), reorder
32
          TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
33
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
34
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
35
          INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
36
          LOGICAL PERIODS(*), REORDER
37
38
          MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian
39
     topology information is attached. If reorder = false then the rank of each process in the
40
     new group is identical to its rank in the old group. Otherwise, the function may reorder
41
     the processes (possibly so as to choose a good embedding of the virtual topology onto
42
     the physical machine). If the total size of the Cartesian grid is smaller than the size of
43
     the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to
44
     MPI_COMM_SPLIT.
45
```

CHAPTER 7. PROCESS TOPOLOGIES

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.

48

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 <i>n</i> -dimensional topology.	
		7 8			
MPI_DIMS_CREATE(nnodes, ndims, dims)					
IN nnodes	number of nodes in a grid (integer)	11			
IN ndims	number of Cartesian dimensions (integer)	12			
INOUT dims	integer array of size ndims specifying the number of	13			
	nodes in each dimension				
<pre>int MPI_Dims_create(int nnodes, int ndims, int dims[])</pre>					
<pre>MPI_Dims_create(nnodes, ndims, dims, ierror) BIND(C) INTEGER, INTENT(IN) :: nnodes, ndims</pre>					
INTEGER, INTENT(IN) :: HHODES, HAIMS INTEGER, INTENT(INOUT) :: dims(ndims) INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) INTEGER NNODES, NDIMS, DIMS(*), IERROR					
			The entries in the array dims are set to describe a Cartesian grid with ndims dimensions $^{2}$		

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.

Negative input values of  $\mathsf{dims}[i]$  are erroneous. An error will occur if  $\mathsf{nnodes}$  is not a multiple of

 $\prod_{i,dims[i]\neq 0} dims[i].$ 

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

293

26

27

28

29 30

 $^{31}$ 

32

33 34 35

36

37 38

#### 7.5.3 Graph Constructor

294

1

2 3 4

5

MPI\_GRAPH\_CREATE(comm\_old, nnodes, index, edges, reorder, comm\_graph)

6	IN	comm_old	input communicator (handle)		
7	IN	nnodes	number of nodes in graph (integer)		
8 9	IN	index	array of integers describing node degrees (see below)		
10	IN	edges	array of integers describing graph edges (see below)		
11	IN	reorder	ranking may be reordered (true) or not (false) (logical)		
12	OUT	comm_graph	communicator with graph topology added (handle)		
13 14					
15	<pre>int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>				
16		const int edges[], i	nt reorder, MPI_Comm *comm_graph)		
17	<pre>MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,</pre>				
18	ierror) BIND(C)				
19	TYPE(MPI_Comm), INTENT(IN) :: comm_old				
20	<pre>INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)</pre>				
21	LOGICAL, INTENT(IN) :: reorder				
22	TYPE(MPI_Comm), INTENT(OUT) :: comm_graph				
23	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror		
24		CDEATE COMM OF NHODES			
25	MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,				
26	IERROR)				
27	<pre>INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR</pre>				
28	LOGICAL REORDER				

MPI\_GRAPH\_CREATE returns a handle to a new communicator to which the graph 2930topology information is attached. If reorder = false then the rank of each process in the 31new group is identical to its rank in the old group. Otherwise, the function may reorder the 32 processes. If the size, nnodes, of the graph is smaller than the size of the group of comm\_old, 33 then some processes are returned MPI\_COMM\_NULL, in analogy to MPI\_CART\_CREATE 34and MPI\_COMM\_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI\_COMM\_NULL 35 is returned in all processes. The call is erroneous if it specifies a graph that is larger than 36 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.

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

46

#### 47 Example 7.2

```
48
```

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

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 edges(j), for  $1 \le j \le$  index(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).

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

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. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  - 1. ndims (number of dimensions),
  - 2. dims (numbers of processes per coordinate direction),
  - 3. periods (periodicity information),
  - 4. own\_position (own position in grid, could also be computed from rank and dims)
- For a graph topology:

index,
 edges,

which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology information. (*End of advice to implementors.*) 45

 $^{24}$ 

# 7.5.4 Distributed Graph Constructor

2 MPI\_GRAPH\_CREATE requires that each process passes the full (global) communication 3 graph to the call. This limits the scalability of this constructor. With the distributed graph 4 interface, the communication graph is specified in a fully distributed fashion. Each process 5specifies only the part of the communication graph of which it is aware. Typically, this 6 could be the set of processes from which the process will eventually receive or get data. 7 or the set of processes to which the process will send or put data, or some combination of 8 such edges. Two different interfaces can be used to create a distributed graph topology. 9 MPI\_DIST\_GRAPH\_CREATE\_ADJACENT creates a distributed graph communicator with 10 each process specifying each of its incoming and outgoing (adjacent) edges in the logical 11 communication graph and thus requires minimal communication during creation. 12

<sup>13</sup> MPI\_DIST\_GRAPH\_CREATE provides full flexibility such that any process can indicate that <sup>14</sup> communication will occur between any pair of processes in the graph.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that 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.

20 21

22

23

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(comm\_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm\_dist\_graph)

23							
24	IN	comm_old	input communicator (handle)				
25 26	IN	indegree	size of sources and sourceweights arrays (non-negative integer)				
			integer)				
27 28	IN	sources	ranks of processes for which the calling process is a				
28 29			destination (array of non-negative integers)				
30	IN	sourceweights	weights of the edges into the calling process (array of				
31		0	non-negative integers)				
32	IN	outdegree	size of destinations and destweights arrays (non-negative				
33		0	integer)				
34	IN	destinations	ranks of processes for which the calling process is a				
35			source (array of non-negative integers)				
36 37	IN	destweights	weights of the edges out of the calling process (array				
38		C C	of non-negative integers)				
39	IN	info	hints on optimization and interpretation of weights				
40			(handle)				
41	IN	reorder	the ranks may be reordered (true) or not (false) (logi-				
42			cal)				
43		comm dist granh	,				
44	OUT	comm_dist_graph	communicator with distributed graph topology (han- dle)				
45			die)				
46							
47	<pre>int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree, const</pre>						
48		<pre>int sources[], const</pre>	<pre>int sourceweights[], int outdegree, const</pre>				

```
1
              int destinations[], const int destweights[], MPI_Info info,
                                                                                   \mathbf{2}
              int reorder, MPI_Comm *comm_dist_graph)
                                                                                   3
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                                                                                   4
              outdegree, destinations, destweights, info, reorder,
                                                                                   5
              comm_dist_graph, ierror) BIND(C)
                                                                                   6
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                   7
    INTEGER, INTENT(IN) ::
                             indegree, sources(indegree), outdegree,
                                                                                   8
    destinations(outdegree)
                                                                                   9
    INTEGER, INTENT(IN) ::
                             sourceweights(*), destweights(*)
                                                                                   10
    TYPE(MPI_Info), INTENT(IN) ::
                                    info
                                                                                   11
    LOGICAL, INTENT(IN) ::
                             reorder
                                                                                   12
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
                                                                                   15
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
                                                                                   16
              OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
                                                                                   17
              COMM_DIST_GRAPH, IERROR)
                                                                                   18
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
                                                                                   19
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
                                                                                   20
    LOGICAL REORDER
```

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about its incoming and outgoing edges in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination 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 topology is the combination of all edges shown in the sources arrays of all processes in comm\_old, which must be identical to the combination of all edges shown in the destinations arrays. Source and destination ranks must be process ranks of comm\_old. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that have specified indegree and outdegree as zero and thus do not occur as source or destination rank in the graph specification) are allowed.

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\_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process 39 remapping strategy and other internal MPI optimizations. For instance, approximate count 40 arguments of later communication calls along specific edges could be used as their edge 41 weights. Multiplicity of edges can likewise indicate more intense communication between 42pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 43 standard and is left to the implementation. In C or Fortran, an application can supply 44the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have 45the same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some 46but not all processes of comm\_old. If the graph is weighted but indegree or outdegree is 47zero, then MPI\_WEIGHTS\_EMPTY or any arbitrary array may be passed to sourceweights 48

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

or destweights respectively. Note that MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are  $\mathbf{2}$ not special weight values; rather they are special values for the total array argument. In 3 Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects like MPI\_BOTTOM (not 4 usable for initialization or assignment). See Section 2.5.4. 5In the case of an empty weights array argument passed while Advice to users. 6 constructing a weighted graph, one should not pass NULL because the value of 7 MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then 8 be indistinguishable from MPI\_UNWEIGHTED to the implementation. In this case 9 MPI\_WEIGHTS\_EMPTY should be used instead. (End of advice to users.) 10 11 Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be imple-12mented as NULL. (End of advice to implementors.) 13 14*Rationale.* To ensure backward compatibility, MPI\_UNWEIGHTED may still be imple-15mented as NULL. See Annex B.1 on page 789. (End of rationale.) 1617The meaning of the info and reorder arguments is defined in the description of the 18 following routine. 192021MPI\_DIST\_GRAPH\_CREATE(comm\_old, n, sources, degrees, destinations, weights, info, reorder, comm\_dist\_graph) 2223IN comm\_old input communicator (handle)  $^{24}$ number of source nodes for which this process specifies IN n 25edges (non-negative integer) 26IN 27sources array containing the n source nodes for which this process specifies edges (array of non-negative integers) 2829IN degrees array specifying the number of destinations for each 30 source node in the source node array (array of non- $^{31}$ negative integers) 32 IN destinations destination nodes for the source nodes in the source 33 node array (array of non-negative integers) 34 IN weights weights for source to destination edges (array of non-35 negative integers) 36 37 IN info hints on optimization and interpretation of weights 38 (handle) 39 IN reorder the process may be reordered (true) or not (false) (log-40 ical) 41 OUT comm\_dist\_graph communicator with distributed graph topology added 42(handle) 43 4445int MPI\_Dist\_graph\_create(MPI\_Comm comm\_old, int n, const int sources[], 46const int degrees[], const int destinations[], const 47int weights[], MPI\_Info info, int reorder, 48 MPI\_Comm \*comm\_dist\_graph)

```
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
             info, reorder, comm_dist_graph, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
    INTEGER, INTENT(IN) :: weights(*)
    TYPE(MPI_Info), INTENT(IN) ::
                                   info
    LOGICAL, INTENT(IN) ::
                            reorder
    TYPE(MPI_Comm), INTENT(OUT) ::
                                    comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
             INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

MPI\_DIST\_GRAPH\_CREATE returns a handle to a new communicator to which the distributed graph topology information is attached. Concretely, each process calls the constructor with a set of directed (source, destination) communication edges as described below. Every process passes an array of n source nodes in the sources array. For each source node, a 20non-negative number of destination nodes is specified in the degrees array. The destination 21nodes are stored in the corresponding consecutive segment of the destinations array. More precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 23j-th such edge stored in destinations[degrees[0]+ $\dots$ +degrees[i-1]+j]. The weight of this edge is stored in weights  $[degrees[0]+\ldots+degrees[i-1]+i]$ . Both the sources and the destinations arrays may contain the same node more than once, and the order in which nodes are listed as destinations or sources is not significant. Similarly, different processes may specify edges with the same source and destination nodes. Source and destination nodes must be process ranks of comm\_old. Different processes may specify different numbers of source and destination nodes, as well as different source to destination edges. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed.

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 is collective.

If reorder = false, all processes will have the same rank in comm\_dist\_graph as in  $comm_old$ . If reorder = true then the MPI library is free to remap to other processes (of comm\_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

42Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count 4344arguments of later communication calls along specific edges could be used as their edge 45weights. Multiplicity of edges can likewise indicate more intense communication between 46pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 47standard and is left to the implementation. In C or Fortran, an application can supply

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9 10

11

12

13

14

1516

17

18

19

22

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some but not all processes of comm\_old. If the graph is weighted but n = 0, then MPI\_WEIGHTS\_EMPTY or any arbitrary array may be passed to weights. Note that MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

- Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI\_UNWEIGHTED to the implementation. In this case MPI\_WEIGHTS\_EMPTY should be used instead. (*End of advice to users.*)
- Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)
- *Rationale.* To ensure backward compatibility, MPI\_UNWEIGHTED may still be implemented as NULL. See Annex B.1 on page 789. (*End of rationale.*)

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.

27 28

29

30

 $^{31}$ 

32

33

34

35

36

37

9

10

11

12

13 14

15

16 17

18

19 20

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.*)

38 39 40

41

**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:

43	process	neighbors
44	0	1, 3
45	1	0
46	2	3
47	3	0, 2
48	L	,

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	0	-	-	-	-
2	0	-	-	-	-
3	0	-	_	_	

In both cases above, the application could supply MPI\_UNWEIGHTED instead of explicitly providing identical weights.

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT could be used to specify this graph using the following arguments:

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	$1,\!3$	1,1	2	$1,\!3$	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	$0,\!2$	1,1	2	0,2	1,1

**Example 7.4** A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI\_DIST\_GRAPH\_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

#### /\*

```
Input: dimensions P, Q
Condition: number of processes equal to P*Q; otherwise only
ranks smaller than P*Q participate
*/
int rank, x, y;
int sources[1], degrees[1];
int destinations[8], weights[8];
MPI_Comm comm_dist_graph;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* get x and y dimension */
y=rank/P; x=rank%P;
```

 $^{31}$ 

```
1
     /* get my communication partners along x dimension */
\mathbf{2}
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
3
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
4
5
     /* get my communication partners along y dimension */
6
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
7
     destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
8
9
     /* get my communication partners along diagonals */
10
     destinations[4] = P*((y+1))(Q)+(x+1)(P); weights[4] = 1;
^{11}
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
12
     destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
13
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
14
15
     sources[0] = rank;
16
     degrees[0] = 8;
17
     MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
18
                             weights, MPI_INFO_NULL, 1, &comm_dist_graph);
19
20
     7.5.5 Topology Inquiry Functions
21
     If a topology has been defined with one of the above functions, then the topology information
22
     can be looked up using inquiry functions. They all are local calls.
23
24
25
     MPI_TOPO_TEST(comm, status)
26
       IN
                                           communicator (handle)
                 comm
27
28
       OUT
                 status
                                            topology type of communicator comm (state)
29
30
     int MPI_Topo_test(MPI_Comm comm, int *status)
^{31}
     MPI_Topo_test(comm, status, ierror) BIND(C)
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         INTEGER, INTENT(OUT) :: status
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
37
          INTEGER COMM, STATUS, IERROR
38
         The function MPI_TOPO_TEST returns the type of topology that is assigned to a
39
     communicator.
40
         The output value status is one of the following:
41
42
       MPI_GRAPH
                                             graph topology
43
       MPI_CART
                                             Cartesian topology
44
       MPI_DIST_GRAPH
                                             distributed graph topology
45
                                             no topology
       MPI_UNDEFINED
46
47
48
```

MPI\_GRAPHDIMS\_GET(comm, nnodes, nedges)

MPI_GRAF	PHDIMS_GET(comm, nnodes,	nedges)	1
IN	comm	communicator for group with graph structure (handle)	2 3
OUT	nnodes	number of nodes in graph (integer) (same as number	4
		of processes in the group)	5
OUT	nedges	number of edges in graph (integer)	6
			7
int MPI_G	raphdims_get(MPI_Comm co	mm, int *nnodes, int *nedges)	8 9
MPI_Graph	dims_get(comm, nnodes, n	edges, ierror) BIND(C)	10
	<pre>MPI_Comm), INTENT(IN) ::</pre>		11
	ER, INTENT(OUT) :: nnod	0	12
	ER, OPTIONAL, INTENT(OUT		13 14
	DIMS_GET(COMM, NNODES, N	-	15
INTEG	ER COMM, NNODES, NEDGES,	LERRUR	16
		and MPI_GRAPH_GET retrieve the graph-topology	17
		communicator by MPI_GRAPH_CREATE. _GRAPHDIMS_GET can be used to dimension the	18 19
	1 U	e following call to MPI_GRAPH_GET.	19 20
			21
	PH_GET(comm, maxindex, ma	vedges index edges)	22
	·	,	23
IN	comm	communicator with graph structure (handle)	24 25
IN	maxindex	length of vector index in the calling program (integer)	26
IN	maxedges	length of vector edges in the calling program	27 28
		(integer)	29
OUT	index	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)	30 31
OUT	edges	array of integers containing the graph structure	32
			33
int MPI_G	raph_get(MPI_Comm comm,	<pre>int maxindex, int maxedges, int index[],</pre>	34 35
	<pre>int edges[])</pre>		36
MPI_Graph	_get(comm, maxindex, max	edges, index, edges, ierror) BIND(C)	37
	<pre>MPI_Comm), INTENT(IN) ::</pre>		38
	ER, INTENT(IN) :: maxin	-	39 40
	ER, OPTIONAL, INTENT(OUT	x(maxindex), edges(maxedges) ) :: jerror	40
			42
		EDGES, INDEX, EDGES, IERROR) GES, INDEX(*), EDGES(*), IERROR	43
111160	LAN OUTT, TRAINDER, TRAED	deb, index(*), iddib(*), iddibit	44
			45 46
			40
			48

```
1
     MPI_CARTDIM_GET(comm, ndims)
2
       IN
                                             communicator with Cartesian structure (handle)
                  comm
3
       OUT
                  ndims
                                             number of dimensions of the Cartesian structure (in-
4
                                             teger)
5
6
7
     int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
8
     MPI_Cartdim_get(comm, ndims, ierror) BIND(C)
9
          TYPE(MPI_Comm), INTENT(IN) :: comm
10
          INTEGER, INTENT(OUT) :: ndims
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
13
14
          INTEGER COMM, NDIMS, IERROR
15
         The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topol-
16
     ogy information that was associated with a communicator by MPI_CART_CREATE. If comm
17
     is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns
18
     ndims=0 and MPI_CART_GET will keep all output arguments unchanged.
19
20
21
     MPI_CART_GET(comm, maxdims, dims, periods, coords)
22
       IN
                  comm
                                             communicator with Cartesian structure (handle)
23
       IN
                  maxdims
                                             length of vectors dims, periods, and
^{24}
                                             coords in the calling program (integer)
25
26
       OUT
                                             number of processes for each Cartesian dimension (ar-
                 dims
27
                                             ray of integer)
28
       OUT
                  periods
                                             periodicity (true/false) for each Cartesian dimension
29
                                             (array of logical)
30
       OUT
                                             coordinates of calling process in Cartesian structure
^{31}
                 coords
                                             (array of integer)
32
33
34
     int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
35
                     int coords[])
36
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) BIND(C)
37
          TYPE(MPI_Comm), INTENT(IN) ::
                                             comm
38
          INTEGER, INTENT(IN) :: maxdims
39
          INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
40
          LOGICAL, INTENT(OUT) :: periods(maxdims)
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
44
          INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
45
          LOGICAL PERIODS(*)
46
47
48
```

MPI\_CART\_RANK(comm, coords, rank) IN communicator with Cartesian structure (handle) comm IN coords integer array (of size ndims) specifying the Cartesian coordinates of a process OUT rank rank of specified process (integer) int MPI\_Cart\_rank(MPI\_Comm comm, const int coords[], int \*rank) MPI\_Cart\_rank(comm, coords, rank, ierror) BIND(C) TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: coords(\*) INTEGER, INTENT(OUT) :: rank INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_CART\_RANK(COMM, COORDS, RANK, IERROR) INTEGER COMM, COORDS(\*), RANK, IERROR For a process group with Cartesian structure, the function MPI\_CART\_RANK translates the logical process coordinates to process ranks as they are used by the point-to-point routines. For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0 or  $coords(i) \ge dims(i)$ , it is shifted back to the interval  $0 \leq coords(i) < dims(i)$  automatically. Out-of-range coordinates are erroneous for nonperiodic dimensions. If comm is associated with a zero-dimensional Cartesian topology, coords is not significant and 0 is returned in rank. MPI\_CART\_COORDS(comm, rank, maxdims, coords) IN communicator with Cartesian structure (handle) comm IN rank of a process within group of comm (integer) rank IN maxdims length of vector **coords** in the calling program (integer) OUT coords integer array (of size ndims) containing the Cartesian coordinates of specified process (array of integers) int MPI\_Cart\_coords(MPI\_Comm comm, int rank, int maxdims, int coords[]) 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)

MPI\_CART\_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
44
INTEGER COMM, RANK, MAXDIMS, COORDS(\*), IERROR
45
46

The inverse mapping, rank-to-coordinates translation is provided by MPI\_CART\_COORDS.

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

1 2

3

4

5 6

7

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

47

```
1
         If comm is associated with a zero-dimensional Cartesian topology,
\mathbf{2}
     coords will be unchanged.
3
4
     MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)
5
6
       IN
                                            communicator with graph topology (handle)
                 comm
7
       IN
                                            rank of process in group of comm (integer)
                 rank
8
       OUT
                 nneighbors
                                            number of neighbors of specified process (integer)
9
10
11
     int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
12
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) BIND(C)
13
          TYPE(MPI_Comm), INTENT(IN) :: comm
14
          INTEGER, INTENT(IN) :: rank
15
          INTEGER, INTENT(OUT) :: nneighbors
16
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
17
18
     MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
19
          INTEGER COMM, RANK, NNEIGHBORS, IERROR
20
21
22
     MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
23
       IN
                                            communicator with graph topology (handle)
                 comm
^{24}
       IN
                 rank
                                            rank of process in group of comm (integer)
25
26
       IN
                 maxneighbors
                                            size of array neighbors (integer)
27
       OUT
                 neighbors
                                            ranks of processes that are neighbors to specified pro-
28
                                             cess (array of integer)
29
30
     int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
^{31}
                    int neighbors[])
32
33
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) BIND(C)
34
          TYPE(MPI_Comm), INTENT(IN) :: comm
35
          INTEGER, INTENT(IN) :: rank, maxneighbors
36
          INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
37
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
39
          INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
40
41
          MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
42
     information for a graph topology. The returned count and array of neighbors for the queried
43
     rank will both include all neighbors and reflect the same edge ordering as was specified by
44
     the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
45
     and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
46
     passed to MPI_GRAPH_CREATE (assuming that index[-1] effectively equals zero):
47
48
```

#### 7.5. TOPOLOGY CONSTRUCTORS

- The 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 argun	ments to MPI_GRAPH_CREATE are:

 $\begin{array}{ll} \text{nnodes} = & 4 \\ \text{index} = & 3, \, 5, \, 6, \, 9 \\ \text{edges} = & 1, \, 1, \, 3, \, 0, \, 0, \, 3, \, 0, \, 2, \, 2 \end{array}$ 

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

Suppose that **comm** is a communicator with a shuffle-exchange topology. The group has  $2^n$  members. Each process is labeled by  $a_1, \ldots, a_n$  with  $a_i \in \{0, 1\}$ , and has three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$  ( $\bar{a} = 1 - a$ ), shuffle $(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$ , and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$ . The graph adjacency list is illustrated below for n = 3.

r	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.

 $^{31}$ 

1 C assume: each process has stored a real number A.  $\mathbf{2}$ C extract neighborhood information 3 CALL MPI\_COMM\_RANK(comm, myrank, ierr) 4 CALL MPI\_GRAPH\_NEIGHBORS(comm, myrank, 3, neighbors, ierr)  $\mathbf{5}$ C perform exchange permutation 6 CALL MPI\_SENDRECV\_REPLACE(A, 1, MPI\_REAL, neighbors(1), 0, 7 neighbors(1), 0, comm, status, ierr) + 8 C perform shuffle permutation 9 CALL MPI\_SENDRECV\_REPLACE(A, 1, MPI\_REAL, neighbors(2), 0, 10 + neighbors(3), 0, comm, status, ierr) 11C perform unshuffle permutation 12CALL MPI\_SENDRECV\_REPLACE(A, 1, MPI\_REAL, neighbors(3), 0, 13 neighbors(2), 0, comm, status, ierr) + 14MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT and MPI\_DIST\_GRAPH\_NEIGHBORS pro-1516vide adjacency information for a distributed graph topology. 1718 MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT(comm, indegree, outdegree, weighted) 19 IN 20communicator with distributed graph topology (hancomm 21dle) 22OUT indegree number of edges into this process (non-negative inte-23ger)  $^{24}$ OUT outdegree number of edges out of this process (non-negative in-25teger) 26OUT weighted false if MPI\_UNWEIGHTED was supplied during cre-27ation, true otherwise (logical) 282930 int MPI\_Dist\_graph\_neighbors\_count(MPI\_Comm comm, int \*indegree,  $^{31}$ int \*outdegree, int \*weighted) 32 MPI\_Dist\_graph\_neighbors\_count(comm, indegree, outdegree, weighted, ierror) 33 BIND(C) 34 TYPE(MPI\_Comm), INTENT(IN) :: comm 35 INTEGER, INTENT(OUT) :: indegree, outdegree 36 LOGICAL, INTENT(OUT) :: weighted 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) 40INTEGER COMM, INDEGREE, OUTDEGREE, IERROR 41 LOGICAL WEIGHTED 4243 444546 4748

•		1 2				
comm	communicator with distributed graph topology (han- dle)	3				
maxindegree	size of sources and sourceweights arrays (non-negative integer)	5 6 7				
sources	processes for which the calling process is a destination (array of non-negative integers)	8 9				
sourceweights	weights of the edges into the calling process (array of non-negative integers)	10 11				
maxoutdegree	size of destinations and destweights arrays (non-negative integer)	12 13 14				
destinations	processes for which the calling process is a source (ar- ray of non-negative integers)	15 16				
destweights	weights of the edges out of the calling process (array of non-negative integers)	17 18 19				
		20 21 22 23				
<pre>MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,</pre>						
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR						
These calls are local. The number of edges into and out of the process returned by 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 (poten- tially 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						
	destinations, destweights comm maxindegree sources sourceweights maxoutdegree destinations destweights Pist_graph_neighbors(MPI_ int sourceweights[], int destweights[]) graph_neighbors(comm, maxoutdegree, destin MPI_Comm), INTENT(IN) :: ER, INTENT(IN) :: maxin ER, INTENT(IN) :: maxin ER, INTENT(OUT) :: sour nations(maxoutdegree) ER :: sourceweights(*), ER, OPTIONAL, INTENT(OUT GRAPH_NEIGHBORS(COMM, MAXOUTDEGREE, DESTIN ER COMM, MAXINDEGREE, SO PESTINATIONS(*), DESTWEIG calls are local. The number _GRAPH_NEIGHBORS_COUN PL_DIST_GRAPH_CREATE_AI processes other than the cal _GRAPH_NEIGHBORS in son	<pre>dle) maxindegree     size of sources and sourceweights arrays (non-negative     integer) sources     processes for which the calling process is a destination     (array of non-negative integers) sourceweights     weights of the edges into the calling process (array of     non-negative integers) maxoutdegree     size of destinations and destweights arrays     (non-negative integers) destinations     processes for which the calling process is a source (ar-     ray of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     weights of the edges out of the calling process (array     of non-negative integers) destweights     int sourceweights[], int maxundegree, int destinations[],     int destweights[]) graph_neighbors(Comm maxindegree, sources, sourceweights,     maxoutdegree) EER, INTENT(IN) :: maxindegree, maxoutdegree EER, INTENT(OUT) :: sourceweights(*) ER, OPTIONAL, INTENT(OUT) :: ierror GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, S</pre>				

MOL DICT COADU NEICUDODC م اہ میں ا iah

sourceweights or destweights or both, or if MPI\_UNWEIGHTED was supplied during the con-4344struction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT then for 4546each 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 4748communicator 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.

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

<sup>13</sup> If the process topology is a Cartesian structure, an MPI\_SENDRECV operation is likely to <sup>15</sup> be used along a coordinate direction to perform a shift of data. As input, MPI\_SENDRECV <sup>16</sup> takes the rank of a source process for the receive, and the rank of a destination process for the <sup>17</sup> send. If the function MPI\_CART\_SHIFT is called for a Cartesian process group, it provides <sup>18</sup> the calling process with the above identifiers, which then can be passed to MPI\_SENDRECV. <sup>19</sup> The user specifies the coordinate direction and the size of the step (positive or negative). <sup>20</sup> The function is local.

```
21
22
```

 $\mathbf{5}$ 

6

7

8

9

10 11 12

MPI\_CART\_SHIFT(comm, direction, disp, rank\_source, rank\_dest)

23	MI I_CAN I		, Talik_Source, Talik_dest)				
24	IN	comm	communicator with Cartesian structure (handle)				
25	IN	direction	coordinate dimension of shift (integer)				
26 27 28	IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)				
28 29	OUT	rank_source	rank of source process (integer)				
30	OUT	rank_dest	rank of destination process (integer)				
31							
32	int MPI_Ca	art_shift(MPI_Comm comm,	int direction, int disp,				
33		<pre>int *rank_source, in</pre>	t *rank_dest)				
34							
35	<pre>MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)</pre>						
36	BIND(C)						
37	TYPE(MPI_Comm), INTENT(IN) :: comm						
38	INTEGER, INTENT(IN) :: direction, disp						
39	INTEGER, INTENT(OUT) :: rank_source, rank_dest						
40	INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
41	MPI_CART_S	SHIFT(COMM, DIRECTION, D	ISP, RANK_SOURCE, RANK_DEST, IERROR)				
42	INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR						
43							
44		0	e coordinate dimension to be traversed by the shift.				
45			ndims-1, where ndims is the number of dimensions.				
46	-		e Cartesian group in the specified coordinate direc-				
47	tion, MPI_C	CARI_SHIFI provides the ide	entifiers for a circular or an end-off shift. In the case				
48							

of an end-off shift, the value MPI\_PROC\_NULL may be returned in rank\_source or rank\_dest, indicating that the source or the destination for the shift is out of range.

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.

#### Example 7.7

The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of REALs is stored one element per process, in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.

```
C find process rank
CALL MPI_COMM_RANK(comm, rank, ierr)
C find Cartesian coordinates
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
+ status, ierr)
```

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 Structures

MPI\_CART\_SUB(comm, remain\_dims, newcomm)

IN	comm	communicator with Cartesian structure (handle)	33	
IN	remain_dims	the i-th entry of remain_dims specifies whether the	34	
IIN	remain_units	· · · ·	35	
		i-th dimension is kept in the subgrid (true) or is drop-	36	
		ped (false) (logical vector)	37	
OUT	newcomm	communicator containing the subgrid that includes	38	
		the calling process (handle)	39	
			40	
int MPI_Ca	art_sub(MPI_Comm comm, co	onst int remain_dims[], MPI_Comm *newcomm)	41	
			42	
	sub(comm, remain_dims, ne	•	43	
	(PI_Comm), INTENT(IN) ::		44	
LOGICA	AL, INTENT(IN) :: remain	n_dims(*)	45	
TYPE(N	TYPE(MPI Comm) INTENT(OUT) · · newcomm			
INTEGE	ER, OPTIONAL, INTENT(OUT)	:: ierror	$46 \\ 47$	
			47	

MPI\_CART\_SUB(COMM, REMAIN\_DIMS, NEWCOMM, IERROR)

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

1 2		R COMM, NEWCOMM, IERROR L REMAIN_DIMS(*)	
3 4 5 6 7 8 9 10	MPI_CART_ form lower-or with the ass comm is alr	SUB can be used to partiti- limensional Cartesian subgrid sociated subgrid Cartesian to eady associated with a zero- with a zero-dimensional Cartes	eated with MPI_CART_CREATE, the function to the communicator group into subgroups that ds, and to build for each subgroup a communicator opology. If all entries in remain_dims are false or dimensional Cartesian topology then newcomm is esian topology. (This function is closely related to
11 12 13 14	<b>Example 7.8</b> Assume that MPI_CART_CREATE(, comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to,		
15	MPI_C	ART_SUB(comm, remain_dim	ns, comm_new),
16 17 18 19 20	will create three communicators each with eight processes in a $2 \times 4$ 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.		
21 22	7.5.8 Low-	-Level Topology Functions	
23 24 25 26 27 28	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.		
29	MPI_CART_	_MAP(comm, ndims, dims, pe	riods, newrank)
30 31	IN	comm	input communicator (handle)
32	IN	ndims	number of dimensions of Cartesian structure (integer)
33 34	IN	dims	integer array of size ndims specifying the number of processes in each coordinate direction
35 36 37	IN	periods	logical array of size ndims specifying the periodicity specification in each coordinate direction
38 39 40 41	OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)
41 42 43	int MPI_Ca	rt_map(MPI_Comm comm, in int periods[], int *1	nt ndims, const int dims[], const newrank)
44 45 46 47 48	<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)</pre>		

	EGER, INTENT(OUT) :: ne EGER, OPTIONAL, INTENT(		1 2	
MDT CAR				
	MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR			
LOGICAL PERIODS(*)			5 6	
	TOGICAT LEWIDD(*)			
	-	optimal" placement for the calling process on the phys-	7	
		tion of this function is to always return the rank of the	8 9	
calling p	calling process, that is, not to perform any reordering.			
Aa	lvice to implementors. Th	ne function MPI_CART_CREATE(comm, ndims, dims,	10 11	
	-	with reorder $=$ true can be implemented by calling	12	
-		s, dims, periods, newrank), then calling	13	
Μ	PI_COMM_SPLIT(comm, cold	or, key, comm_cart), with color = 0 if newrank $ eq$	14	
Μ	$PI\_UNDEFINED,  color = MP$	I_UNDEFINED otherwise, and $key = newrank$ . If ndims	15	
is	zero then a zero-dimensional	Cartesian topology is created.	16	
Tł	he function $MPI\_CART\_SUB$	(comm, remain_dims, comm_new) can be implemented	17	
by	a call to MPI_COMM_SPLI	Γ(comm, color, key, comm_new), using a single number	18	
en	coding of the lost dimensions	as <b>color</b> and a single number encoding of the preserved	19	
di	mensions as key.		20	
Al	l other Cartesian topology fu	nctions can be implemented locally, using the topology	21	
int	formation that is cached with	the communicator. (End of advice to implementors.)	22	
<b>T</b> -			23 24	
1 116	corresponding function for	graph structures is as follows.	24 25	
			26	
MPI_GR	APH_MAP(comm, nnodes, in	ndex, edges, newrank)	27	
IN	comm	input communicator (handle)	28	
IN	nnodes	number of graph nodes (integer)	29	
IN	index	integer array specifying the graph structure, see	30	
	index	MPI_GRAPH_CREATE	31 32	
IN	edges	integer array specifying the graph structure	33	
OUT	newrank	reordered rank of the calling process;	34	
001	newrank	MPI_UNDEFINED if the calling process does not be-	35	
		long to graph (integer)	36	
			37	
int MPI	Graph map(MPT Comm comm	n, int nnodes, const int index[], const	38	
	int edges[], int		39	
	C		40	
		<pre>dex, edges, newrank, ierror) BIND(C)</pre>	41	
	PE(MPI_Comm), INTENT(IN)		42	
	EGER, INTENI(IN) :: nno EGER, INTENT(OUT) :: no	odes, index(nnodes), edges(*)	43 44	
	EGER, INTENI(UOI) :: ne EGER, OPTIONAL, INTENT((		44 45	
			40	
		DEX, EDGES, NEWRANK, IERROR)	47	
INT	EGER COMM, NNODES, INDEX	K(*), EDGES(*), NEWRANK, IERROR	48	

Advice to implementors. The function MPI\_GRAPH\_CREATE(comm, nnodes, index, edges, reorder, comm\_graph), with reorder = true can be implemented by calling MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank), then calling MPI\_COMM\_SPLIT(comm, color, key, comm\_graph), with color = 0 if newrank ≠ 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.*)

## 7.6 Neighborhood Collective Communication on Process Topologies

<sup>12</sup> MPI process topologies specify a communication graph, but they implement no commu-<sup>13</sup> nication function themselves. Many applications require sparse nearest neighbor commu-<sup>14</sup> nications that can be expressed as graph topologies. We now describe several collective <sup>15</sup> operations that perform communication along the edges of a process topology. All of these <sup>16</sup> functions are collective; i.e., they must be called by all processes in the specified com-<sup>17</sup> municator. See Section 5 on page 141 for an overview of other dense (global) collective <sup>18</sup> communication operations and the semantics of collective operations.

<sup>19</sup> If the graph was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT with sources <sup>20</sup> and destinations containing 0, ..., n-1, where n is the number of processes in the group <sup>21</sup> of comm\_old (i.e., the graph is fully connected and also includes an edge from each node <sup>22</sup> to itself), then the sparse neighborhood communication routine performs the same data <sup>23</sup> exchange as the corresponding dense (fully-connected) collective operation. In the case of a <sup>24</sup> Cartesian communicator, only nearest neighbor communication is provided, corresponding <sup>25</sup> 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.)

<sup>33</sup> For a distributed graph topology, created with MPI\_DIST\_GRAPH\_CREATE, the se-<sup>34</sup> quence of neighbors in the send and receive buffers at each process is defined as the se-<sup>35</sup> quence returned by MPI\_DIST\_GRAPH\_NEIGHBORS for destinations and sources, respec-<sup>36</sup> tively. For a general graph topology, created with MPI\_GRAPH\_CREATE, the order of <sup>37</sup> neighbors in the send and receive buffers is defined as the sequence of neighbors as re-<sup>38</sup> turned by MPI\_GRAPH\_NEIGHBORS. Note that general graph topologies should generally <sup>39</sup> 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.

<sup>47</sup> If a neighbor in any of the functions is MPI\_PROC\_NULL, then the neighborhood collec-<sup>48</sup> tive communication behaves like a point-to-point communication with MPI\_PROC\_NULL in

1

2

3

4

5

6

7

8 9

10 11

26

27

28

29

30

31

this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

#### Neighborhood Gather 7.6.1

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block in the receive buffer is received from the *l*-th neighbor.

MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,			11
comm)			12
	commy		13
IN	sendbuf	starting address of send buffer (choice)	14
IN	sendcount	number of elements sent to each neighbor (non-negative	15
		integer)	16
IN	sendtype	data type of send buffer elements (handle)	17
IIN	senatype	data type of send buller elements (nandle)	18
OUT	recvbuf	starting address of receive buffer (choice)	19
IN	recvcount	number of elements received from each neighbor (non-	20
		negative integer)	21
IN	recvtype	data type of receive buffer elements (handle)	22
IIN	Тестуре	data type of receive buller elements (nandle)	$^{23}$
IN	comm	communicator with topology structure (handle)	24
			25
int MPI_Ne	eighbor_allgather(const v	oid* sendbuf, int sendcount, MPI_Datatype	26
	sendtype, void* recvb	ouf, int recvcount, MPI_Datatype recvtype,	27
	MPI_Comm comm)		28

29MPI\_Neighbor\_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 30 recvtype, comm, ierror) BIND(C) TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 32 TYPE(\*), DIMENSION(..) :: recvbuf 33 INTEGER, INTENT(IN) :: sendcount, recvcount 34 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 35 TYPE(MPI\_Comm), INTENT(IN) :: comm 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI\_NEIGHBOR\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 38 RECVTYPE, COMM, IERROR) 39 <type> SENDBUF(\*), RECVBUF(\*)

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph 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);

1

 $\mathbf{2}$ 

3 4

5

6

7

8

9 10 1

31

40

41 42

43

44

45

4647

```
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);
\mathbf{5}
      int k,l;
6
\overline{7}
      /* assume sendbuf and recvbuf are of type (char*) */
8
      for(k=0; k<outdegree; ++k)</pre>
9
        MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
10
11
      for(l=0; l<indegree; ++1)</pre>
12
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
13
                     srcs[1]...);
14
15
     MPI_Waitall(...);
16
           Figure 7.1 shows the neighborhood gather communication of one process with outgoing
17
      neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf to
18
      all four destinations (outgoing neighbors) and it will receive the contribution from all six
19
      sources (incoming neighbors) into separate locations of its receive buffer.
20
21
                                                d_0
22
                                                                d_2, s_4
23
                                             s_0
24
25
                                d_1
                                                            s_1
26
27
28
                                                                    s_3
                                             s_2
29
30
                                                        d_{3}, s_{5}
^{31}
                        sendbuf
32
33
34
                                             s_1
                                                                             s_5
                                     s_0
                                                     s_2
                                                             s_3
                                                                     s_4
```

Figure 7.1: FIXME: You cannot use the label command without a caption

All arguments are significant on all processes and the argument comm must have identical values on all processes.

recvbuf

The type signature associated with sendcount, sendtype, at a process must be equal to 42the type signature associated with recvcount, recvtype at all other processes. This implies 43 that the amount of data sent must be equal to the amount of data received, pairwise between 44 every pair of communicating processes. Distinct type maps between sender and receiver are 45 still allowed. 46

47Rationale. For optimization reasons, the same type signature is required indepen-48 dently of whether the topology graph is connected or not. (*End of rationale.*)

1

35

36 37

38 39

40

#### 7.6. NEIGHBORHOOD COLLECTIVE COMMUNICATION

1 The "in place" option is not meaningful for this operation. The vector variant of MPI\_NEIGHBOR\_ALLGATHER allows one to gather different 2 3 numbers of elements from each neighbor. 4 5MPI\_NEIGHBOR\_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 6 recvtype, comm) 7 8 IN sendbuf starting address of send buffer (choice) 9 IN sendcount number of elements sent to each neighbor (non-negative 10 integer) 11 IN sendtype data type of send buffer elements (handle) 1213 OUT recvbuf starting address of receive buffer (choice) 14non-negative integer array (of length indegree) con-IN recvcounts 15taining the number of elements that are received from 16 each neighbor 17IN displs integer array (of length indegree). Entry i specifies the 18 displacement (relative to recvbuf) at which to place the 19 incoming data from neighbor i 2021IN data type of receive buffer elements (handle) recvtype 22 IN communicator with topology structure (handle) comm 23 $^{24}$ int MPI\_Neighbor\_allgatherv(const void\* sendbuf, int sendcount, 25MPI\_Datatype sendtype, void\* recvbuf, const int recvcounts[], 26const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm) 2728 MPI\_Neighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 29 displs, recvtype, comm, ierror) BIND(C) 30 TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 31TYPE(\*), DIMENSION(..) :: recvbuf 32 INTEGER, INTENT(IN) :: sendcount, recvcounts(\*), displs(\*) 33 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(MPI\_Comm), INTENT(IN) :: comm 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI\_NEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 37 DISPLS, RECVTYPE, COMM, IERROR) 38

<type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, IERROR

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph 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)); 317

39

40

 $41 \\ 42$ 

43

44

45

 $46 \\ 47$ 

```
1
     int *dsts=(int*)malloc(outdegree*sizeof(int));
^{2}
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
3
                                outdegree,dsts,MPI_UNWEIGHTED);
4
     int k,l;
5
6
     /* assume sendbuf and recvbuf are of type (char*) */
7
     for(k=0; k<outdegree; ++k)</pre>
8
       MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
9
10
     for(l=0; l<indegree; ++1)</pre>
11
       MPI_Irecv(recvbuf+displs[l]*extent(recvtype),recvcounts[l],recvtype,
12
                  srcs[1],...);
```

```
13 \\ 14
```

15

MPI\_Waitall(...);

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[I], recvtype at any other process with srcs[I]==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 data received from the I-th neighbor is placed into recvbuf beginning at offset displs[I] elements (in terms of the recvtype).

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.

 $25 \\ 26$ 

22

## 7.6.2 Neighbor Alltoall

In this function, each process i receives data items from each process j if an edge (j,i)exists in the topology graph or Cartesian topology. Similarly, each process i sends data items to all processes j where an edge (i, j) exists. This call is more general than

<sup>31</sup> MPI\_NEIGHBOR\_ALLGATHER in that different data items can be sent to each neighbor. <sup>32</sup> The *k*-th block in send buffer is sent to the *k*-th neighboring process and the *l*-th block in <sup>33</sup> the receive buffer is received from the *l*-th neighbor.

34

- 35 36
- 37
- 38
- 39 40
- 41
- 42

43

- 44
- 45
- $46 \\ 47$

MPI_NE	EIGHBOR_ALLTOALL(sei comm)	ndbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	1 $2$
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative	4 5
	Sendeoune	integer)	6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	starting address of receive buffer (choice)	8 9
IN	recvcount	number of elements received from each neighbor (non-negative integer)	9 10 11
IN	recvtype	data type of receive buffer elements (handle)	12
IN	comm	communicator with topology structure (handle)	13 14
int MP	-	const void* sendbuf, int sendcount, MPI_Datatype d* recvbuf, int recvcount, MPI_Datatype recvtype, )	15 16 17 18
MPI_Ne:	-	ouf, sendcount, sendtype, recvbuf, recvcount, n, ierror) BIND(C)	19 20
TYI	• •	INTENT(IN) :: sendbuf	21
	PE(*), DIMENSION()		22 23
		sendcount, recvcount	23 24
	PE(MPI_Datatype), INI PE(MPI_Comm), INTENT(	<pre>CENT(IN) :: sendtype, recvtype (IN) :: comm</pre>	25
	TEGER, OPTIONAL, INTE		26
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>			27 28 29 30
IN	TEGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	31
		esian communicators, graph communicators, and distributed	32
		ed in Section 7.6 on page 314. If comm is a distributed graph	33 34
		as if each process executed sends to each of its outgoing h of its incoming neighbors:	35
neignoo	is and receives from each	n of its incoming neighbors.	36
MPI_Di	st_graph_neighbors_cc	<pre>ount(comm,&amp;indegree,&amp;outdegree,&amp;weighted);</pre>	37
	rcs=(int*)malloc(inde	-	38
	sts=(int*)malloc(outd	-	39
MPI_Di:		<pre>mm,indegree,srcs,MPI_UNWEIGHTED,</pre>	40 41
int k,		<pre>itdegree,dsts,MPI_UNWEIGHTED);</pre>	41
тно к,.	⊥,		43
/* assi	ume sendbuf and recvb	ouf are of type (char*) */	44
	0; k <outdegree; ++k)<="" td=""><td></td><td>45</td></outdegree;>		45
MPI_		count*extent(sendtype),sendcount,sendtype,	46
	dsts[k],);		47
			48

```
1
     for(l=0; l<indegree; ++1)</pre>
\mathbf{2}
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
3
                    srcs[1],...);
4
5
     MPI_Waitall(...);
6
          The type signature associated with sendcount, sendtype, at a process must be equal to
7
      the type signature associated with recvcount, recvtype at any other process. This implies
8
      that the amount of data sent must be equal to the amount of data received, pairwise between
9
      every pair of communicating processes. Distinct type maps between sender and receiver are
10
     still allowed.
11
          The "in place" option is not meaningful for this operation.
12
          All arguments are significant on all processes and the argument
13
      comm must have identical values on all processes.
14
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
15
      numbers of elements to and from each neighbor.
16
17
18
      MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
19
                      rdispls, recvtype, comm)
20
                  sendbuf
       IN
                                                starting address of send buffer (choice)
21
22
       IN
                  sendcounts
                                                non-negative integer array (of length outdegree) speci-
23
                                                fying the number of elements to send to each neighbor
^{24}
       IN
                                                integer array (of length outdegree). Entry j specifies
                  sdispls
25
                                                the displacement (relative to sendbuf) from which to
26
                                                send the outgoing data to neighbor j
27
       IN
                  sendtype
                                                data type of send buffer elements (handle)
28
29
                  recvbuf
       OUT
                                                starting address of receive buffer (choice)
30
       IN
                  recvcounts
                                                non-negative integer array (of length indegree) speci-
^{31}
                                                fying the number of elements that are received from
32
                                                each neighbor
33
       IN
                  rdispls
                                                integer array (of length indegree). Entry i specifies the
34
                                                displacement (relative to recvbuf) at which to place the
35
                                                incoming data from neighbor i
36
37
       IN
                                                data type of receive buffer elements (handle)
                  recvtype
38
       IN
                  comm
                                                communicator with topology structure (handle)
39
40
      int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],
41
                      const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
42
                      const int recvcounts[], const int rdispls[], MPI_Datatype
43
                      recvtype, MPI_Comm comm)
44
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
```

```
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_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                10
    RECVTYPE, COMM, IERROR
                                                                               11
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph 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;

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)</pre>
```

```
MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
          dsts[k],...);
```

```
for(l=0; l<indegree; ++1)</pre>
 MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype),recvcounts[1],recvtype,
            srcs[1],...);
```

```
MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process 35i must be equal to the type signature associated with recvcounts[I], recvtype with srcs[I]==i 36 at process j. This implies that the amount of data sent must be equal to the amount of 37 data received, pairwise between every pair of communicating processes. Distinct type maps 38 between sender and receiver are still allowed. The data in the sendbuf beginning at offset 39 sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data 40 received from the I-th incoming neighbor is placed into recvbuf beginning at offset rdispls[I] 41 elements (in terms of the recvtype). 42

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.

MPI\_NEIGHBOR\_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

1

2

3

4

5

6

7

8

9

12

13 14

15

16

1718

19

20

21

22 23

 $^{24}$ 

25

26

2728

29

30

3132

33 34

43

44

45

46

1 2	MPI_NEI	GHBOR_ALLTOALLW(sen rdispls, recvtypes, c	ndbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, comm)
$\frac{3}{4}$	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	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)
14 15	OUT	recvbuf	starting address of receive buffer (choice)
16 17 18	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
19 20 21 22	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)
23 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 28	IN	comm	communicator with topology structure (handle)
29 30 31 32 33	int MPI_	const MPI_Aint s void* recvbuf, d	onst void* sendbuf, const int sendcounts[], sdispls[], const MPI_Datatype sendtypes[], const int recvcounts[], const MPI_Aint t MPI_Datatype recvtypes[], MPI_Comm comm)
34 35 36 37 38 39 40 41 42	TYPE TYPE INTE INTE TYPE TYPE	recvcounts, rdis (*), DIMENSION(), I (*), DIMENSION() : GER, INTENT(IN) :: s GER(KIND=MPI_ADDRESS_	: recvbuf sendcounts(*), recvcounts(*) KIND), INTENT(IN) :: sdispls(*), rdispls(*) NT(IN) :: sendtypes(*), recvtypes(*) N) :: comm
43 44 45 46 47 48	<tyr INTE</tyr 	RECVCOUNTS, RDIS pe> SENDBUF(*), RECVE GER(KIND=MPI_ADDRESS_ GER SENDCOUNTS(*), SE	UF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, SPLS, RECVTYPES, COMM, IERROR) BUF(*) KIND) SDISPLS(*), RDISPLS(*) ENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
for(k=0; k<outdegree; ++k)
MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);</pre>
```

```
for(1=0; 1<indegree; ++1)
MPI_Irecv(recvbuf+rdispls[1],recvcounts[1], recvtypes[1],srcs[1],...);</pre>
```

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[l], recvtypes[l] with srcs[l]==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.

## 7.7 Nonblocking Neighborhood Communication on Process Topologies

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.  $\mathbf{2}$ 

 $^{24}$ 

	324		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.7.1 No	nblocking Neighborhood Gatł	ıer
4 5 6	MPI_INEIG	GHBOR_ALLGATHER(sendbuf, comm, request)	sendcount, sendtype, recvbuf, recvcount, recvtype,
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 12	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)
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	MPI_Ineig TYPE( TYPE( INTEG TYPE( TYPE( INTEG MPI_INEIG ( INTEG	<pre>MPI_Datatype sendtype MPI_Datatype recvtype mPI_Datatype recvtype mecvtype, comm, reque (*), DIMENSION(), INTENT (*), DIMENSION(), ASYNCH ER, INTENT(IN) :: sendco MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) ER, OPTIONAL, INTENT(OUT) HBOR_ALLGATHER(SENDBUF, S RECVTYPE, COMM, REQUE ENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, F</pre>	<pre>T(IN), ASYNCHRONOUS :: sendbuf HRONOUS :: recvbuf ount, recvcount ) :: sendtype, recvtype comm ) :: request ) :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, EST, IERROR)</pre>

MPI_INE	EIGHBOR_ALLGATHERV( recvtype, comm, re	sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, equest)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor	9 10 11 12
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	13 14 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
TYP TYP INT INT TYP TYP INT MPI_INE	const int displ MPI_Request *re displs, recvtyp E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: EGER, INTENT(IN), ASY E(MPI_Datatype), INTE E(MPI_Comm), INTENT(I E(MPI_Request), INTENT EGER, OPTIONAL, INTENT EGER, OPTIONAL, INTENT CIGHBOR_ALLGATHERV(SENT DISPLS, RECVTYP	<pre>dbuf, sendcount, sendtype, recvbuf, recvcounts, e, comm, request, ierror) BIND(C) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount NCHRONOUS :: recvcounts(*), displs(*) NT(IN) :: sendtype, recvtype N) :: comm T(OUT) :: request T(OUT) :: request T(OUT) :: ierror DBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, E, COMM, REQUEST, IERROR)</pre>	22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
INT REQ	UEST, IERROR	BUF(*) YPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, g variant of MPI_NEIGHBOR_ALLGATHERV.	39 40 41 42 43 44
			45

	326		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.7.2 No	nblocking Neighborhood Allte	oall
4 5	MPI_INEIG	GHBOR_ALLTOALL(sendbuf, s comm, request)	endcount, sendtype, recvbuf, recvcount, recvtype,
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 12	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)
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	TYPE( TYPE( INTEG TYPE( TYPE( TYPE( INTEG MPI_INEIG <type INTEG</type 	<pre>MPI_Comm comm, MPI_R hbor_alltoall(sendbuf, se recvtype, comm, requ *), DIMENSION(), INTENT *), DIMENSION(), ASYNCH ER, INTENT(IN) :: sendce MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) HBOR_ALLTOALL(SENDBUF, SI RECVTYPE, COMM, REQU e&gt; SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, INTENT(PARCE)</pre>	<pre>endcount, sendtype, recvbuf, recvcount, est, ierror) BIND(C) T(IN), ASYNCHRONOUS :: sendbuf HRONOUS :: recvbuf ount, recvcount ) :: sendtype, recvtype comm ) :: request ) :: ierror ENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, EST, IERROR)</pre>
40 41 42 43 44 45 46 47 48			

MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, <sup>1</sup> rdispls, recvtype, comm, request) <sup>2</sup>				
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	4 5 6	
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j	7 8 9	
IN	sendtype	data type of send buffer elements (handle)	10 11	
OUT	recvbuf	starting address of receive buffer (choice)	12	
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	13 14 15	
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	16 17 18 19	
IN	recvtype	data type of receive buffer elements (handle)	20	
IN	comm	communicator with topology structure (handle)	21	
OUT	request	communication request (handle)	22 23	
			24	
<pre>int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>				
	const int recvcount	comm, MPI_Request *request)	26 27 28	
MPT Inei	whor alltoally(sendbuf.	sendcounts, sdispls, sendtype, recvbuf,	29	
		s, recvtype, comm, request, ierror) BIND(C)	30	
	(*), DIMENSION(), INTE		31 32	
	(*), DIMENSION(), ASYN	CHRONOUS :: recvbuf ONOUS :: sendcounts(*), sdispls(*),	33	
	counts(*), rdispls(*)	UNUUS Senacounts(*), Satspis(*),	34	
	• •	N) :: sendtype, recvtype	35	
	(MPI_Comm), INTENT(IN) :		36 37	
	(MPI_Request), INTENT(OU GER, OPTIONAL, INTENT(OU	-	38	
			39	
MPI_INEI		SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, S, RECVTYPE, COMM, REQUEST, IERROR)	40	
<tvr< td=""><td>e&gt; SENDBUF(*), RECVBUF(</td><td></td><td>41 42</td></tvr<>	e> SENDBUF(*), RECVBUF(		41 42	
	-	LS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	42 43	
RECV	IYPE, COMM, REQUEST, IER	ROR	44	
This	call starts a nonblocking vari	iant of MPI_NEIGHBOR_ALLTOALLV.	45	
	-		46	
47				

1 2	MPI_INEI	GHBOR_ALLTOALLW(sendb rdispls, recvtypes, com	ouf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, im, request)
$\frac{3}{4}$	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) speci- fying the number of elements that 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)
30 31 32 33 34 35	<pre>int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>		
36	MPI_Ineig	ghbor_alltoallw(sendbuf	, sendcounts, sdispls, sendtypes, recvbuf,
37			s, recvtypes, comm, request, ierror) BIND(C)
38 39			ENT(IN), ASYNCHRONOUS :: sendbuf
40		(*), DIMENSION(), ASY SEB INTENT(IN) ASYNCH	RONOUS :: recvour RONOUS :: sendcounts(*), recvcounts(*)
41			ND), INTENT(IN), ASYNCHRONOUS ::
42		pls(*), rdispls(*)	
43		• -	<pre>IN), ASYNCHRONOUS :: sendtypes(*),</pre>
44		types(*)	
45 46		(MPI_Comm), INTENT(IN)	
47		(MPI_Request), INTENT(O GER, OPTIONAL, INTENT(O	-
48	1111		

This call starts a nonblocking variant of MPI\_NEIGHBOR\_ALLTOALLW.

## 7.8 An Application Example

**Example 7.9** The example in 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 the points u(1:100,1:100) owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the 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)).

 $\mathbf{2}$ 

```
2
3
4
5
6
7
8
     INTEGER ndims, num_neigh
9
     LOGICAL reorder
10
     PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
11
     INTEGER comm, comm_cart, dims(ndims), ierr
12
     INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
13
     LOGICAL periods(ndims)
14
     REAL u(0:101,0:101), f(0:101,0:101)
15
     DATA dims / ndims * 0 /
16
     comm = MPI_COMM_WORLD
17
     !
         Set process grid size and periodicity
18
     CALL MPI_DIMS_CREATE(comm, ndims, dims,ierr)
19
     periods(1) = .TRUE.
20
     periods(2) = .TRUE.
21
         Create a grid structure in WORLD group and inquire about own position
22
     CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
23
                        comm_cart,ierr)
^{24}
     CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords,ierr)
25
     i = own_coords(1)
26
     j = own_coords(2)
27
     ! Look up the ranks for the neighbors. Own process coordinates are (i,j).
28
     ! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
     CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
29
30
     CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3),neigh_rank(4), ierr)
^{31}
     ! Initialize the grid functions and start the iteration
32
     CALL init (u, f)
33
     DO it=1,100
34
        CALL relax (u, f)
35
     !
            Exchange data with neighbor processes
36
        CALL exchange (u, comm_cart, neigh_rank, num_neigh)
37
     END DO
38
     CALL output (u)
39
40
        Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
41
42
43
44
45
46
47
48
```

```
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
                                                                                   12
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                   13
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
                                                                                   14
INTEGER ierr
sndbuf(1:100,1) = u( 1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100, 1)
                                                                                   18
sndbuf(1:100,4) = u(1:100,100)
                                                                                   19
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                                                                                   20
                             comm_cart, ierr)
                                                                                   21
! instead of
                                                                                   22
! DO i=1,num_neigh
                                                                                   23
    CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),&
!
                                                                                   24
!
                    ierr)
                                                                                   25
!
    CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i),&
                                                                                   26
!
                    ierr)
                                                                                   27
! END DO
                                                                                   28
! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
                                                                                   29
                                                                                   30
u( 0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
                                                                                   32
u(1:100, 0) = rcvbuf(1:100,3)
                                                                                   33
u(1:100,101) = rcvbuf(1:100,4)
                                                                                   34
END
                                                                                   35
                                                                                   36
```

Figure 7.3: Communication routine with local data copying and sparse neighborhood allto-all.

11

15

16

17

 $^{31}$ 

37

38

```
1
\mathbf{2}
3
     SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
4
     USE MPI
\mathbf{5}
    REAL u(0:101,0:101)
6
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
7
     INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
8
     INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
9
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal, sdispls(num_neigh), &
10
                                      rdispls(num_neigh)
11
     INTEGER type_vec, i, ierr
12
         The following initialization need to be done only once
13
     !
         before the first call of exchange.
14
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
15
     CALL MPI_TYPE_VECTOR (100, 1, 102, MPI_REAL, type_vec, ierr)
16
     CALL MPI_TYPE_COMMIT (type_vec, ierr)
17
     sndtypes(1) = type_vec
18
     sndtypes(2) = type_vec
19
     sndtypes(3) = MPI_REAL
20
     sndtypes(4) = MPI_REAL
21
    DO i=1,num_neigh
22
        sndcounts(i) = 100
23
        rcvcounts(i) = 100
^{24}
        rcvtypes(i) = sndtypes(i)
25
     END DO
26
     sdispls(1) = ( 1 + 1*102) * sizeofreal
                                                   ! first element of u( 1,1:100)
     sdispls(2) = (100 + 1*102) * sizeofreal
27
                                                   ! first element of u(100,1:100)
28
     sdispls(3) = ( 1 + 1*102) * sizeofreal
                                                  ! first element of u(1:100, 1)
29
     sdispls(4) = ( 1 + 100*102) * sizeofreal ! first element of u(1:100,100)
30
    rdispls(1) = (0 + 1*102) * size of real ! first element of u(0,1:100)
^{31}
    rdispls(2) = (101 + 1*102) * sizeofreal
                                                  ! first element of u(101,1:100)
32
     rdispls(3) = (1 + 0*102) * size of real ! first element of u(1:100, 0)
33
    rdispls(4) = ( 1 + 101*102) * sizeofreal ! first element of u(1:100,101)
34
35
     ! the following communication has to be done in each call of exchange
36
     CALL MPI_NEIGHBOR_ALLTOALLW (u, sndcounts, sdispls, sndtypes, &
37
                                 u, rcvcounts, rdispls, rcvtypes, comm_cart, ierr)
38
39
     !
         The following finalizing need to be done only once
40
     !
         after the last call of exchange.
41
     CALL MPI_TYPE_FREE (type_vec, ierr)
42
     END
43
44
     Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local
45
     data copying.
46
47
48
```

# 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.

 $^{24}$ 

#### Implementation Information 8.1

### 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,

#define MPI\_VERSION #define MPI\_SUBVERSION 0

in Fortran,		
III I OI OI dail,		
INTEGER :: MPI_VERSION, MPI_SUBVERSION		
PARAMETER (MPI_VERSION = 3)	)	34
PARAMETER (MPI_SUBVERSION = 0)	)	35
		36
For runtime determination,		37
		38
MPI_GET_VERSION( version, subversion	)	39
	)	40
OUT version	version number (integer)	41
OUT subversion	subversion number (integer)	42
		43
<pre>int MPI_Get_version(int *version,</pre>	int *subversion)	44
,		45
MPI_Get_version(version, subversion)	on, ierror) BIND(C)	46
INTEGER, INTENT(OUT) :: versi	ion, subversion	47
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	48

	004		UIAI IER 8	. WIFTENVINO		AGEMEN I
1	MPT GET	_VERSION(VERSIO	N. SUBVERSION.	TERROR)		
2		EGER VERSION, SU				
3	MPI	GET VERSION of	an be called befo	ore MPI INIT an	d after MPI_FINA	I IZE. Valid
4 5					versions of the MI	
6	are $(3,0)$ ,	(2,2), (2,1), (2,0)	, and $(1,2)$ .	-		
7						
8 9	MPI_GE1	LIBRARY_VERS	SION( version, resu	ıltlen )		
9 10	OUT	version	ver	rsion string (string	;)	
11	OUT	resultlen	Le	ngth (in printable	characters) of the res	sult returned
12			in	version (integer)		
13 14	int MPI_	_Get_library_ve	rsion(char *ver	sion, int *res	sultlen)	
15 16	MPI_Get_	_library_version	n(version, resu	lten, ierror)	BIND(C)	
17	CHAF	RACTER (LEN=MPI_1	MAX_LIBRARY_VER	SION_STRING),	INTENT(OUT) ::	version
18		EGER, INTENT(OUT				
19	INTE	EGER, OPTIONAL,	INTENT(OUT) ::	ierror		
20 21		LIBRARY_VERSIO	-	LTEN, IERROR)		
21		ACTER*(*) VERS				
23		EGER RESULTLEN,	LERKUK			
24 25		routine returns a t is a character str	· ·	0	the MPI library.	The version
26	Ada	vice to implemente	ors. An impleme	ntation of MPI s	hould return a diffe	erent string
27 28		-	-		be visible to the us	0
29	adi	vice to implemento	rs.)			
30	The	argument version	must represent st	orage that is		
31 32		0	*	0	GET_LIBRARY_VE	RSION may
33	-	to this many char				
34					the output argumen	
35					]. The value of resu Fortran, version is	
36	0				not be larger than	•
37	0	LIBRARY_VERSIC			dia	
$\frac{38}{39}$			/ERSION can be	called before $MF$	$PI\_INIT$ and after	
40	MPI_FIN	ALIZE.				
41	010 F					
42	8.1.2 E	nvironmental Inqu	lines			
43					are attached to t	
44 45					values of these att: $R$ described in Sec	
46	-				to delete these attr	
47		s, or change their				
48	•	list of predefined		ludo		

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

<sup>48</sup> The list of predefined attribute keys include

MPI\_PROC\_NULL will be returned.

<b>MPI_TAG_UB</b> Upper bound for tag value.	1
<b>MPI_HOST</b> Host process rank, if such exists, MPI_PROC_NULL, otherwise.	2 3
<b>MPI_IO</b> rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.	4 5 6
<b>MPI_WTIME_IS_GLOBAL</b> Boolean variable that indicates whether clocks are synchronized.	7
Vendors may add implementation-specific parameters (such as node number, real mem- ory size, virtual memory size, etc.) These predefined attributes do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	8 9 10 11 12
Advice to users. Note that in the C binding, the value returned by these attributes is a <i>pointer</i> to an <b>int</b> containing the requested value. ( <i>End of advice to users.</i> )	13 14 15
The required parameter values are discussed in more detail below:	$16 \\ 17$
Tag Values Tag values range from 0 to the value returned for MPI_TAG_UB, inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be <i>at least</i> 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a valid value	18 19 20 21 22 23
for MPI_TAG_UB. The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD. Host Rank	24 25 26 27
The value returned for MPI_HOST gets the rank of the <i>HOST</i> process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a <i>HOST</i> , nor does it requires that a <i>HOST</i> exists. The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	28 29 30 31 32 33
IO Rank	$34 \\ 35$
The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ISO C I/O operations are supported (e.g., fopen, fprintf, lseek).	36 37 38 39 40
If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value	40 41 42 43 44

Advice to users. Note that input is not collective, and this attribute does not indicate which process can or does provide input. (End of advice to users.)

45

46

47

1 2 3 4 5 6 7 8 9 10 11 12 13 14	The value MPI_COM synchronia the variat round-trip before a s be always The synchronia attribute The	M_WORLD are synchronized, ized if explicit effort has been to tion in time, as measured by or p time for an MPI message of send and at another process ju s higher than the first one. attribute MPI_WTIME_IS_GLO ized (however, the attribute k may be associated with comm	GLOBAL is 1 if clocks at all processes in 0 otherwise. A collection of clocks is considered taken to synchronize them. The expectation is that alls to MPI_WTIME, will be less then one half the length zero. If time is measured at a process just st after a matching receive, the second time should BAL need not be present when the clocks are not ey MPI_WTIME_IS_GLOBAL is always valid). This unicators other then MPI_COMM_WORLD. BAL has the same value on all processes of
15 16 17	Inquire Pr	rocessor Name	
18 19 20		_PROCESSOR_NAME( name,	,
21	OUT	name	A unique specifier for the actual (as opposed to vir- tual) node.
22 23 24	OUT	resultlen	Length (in printable characters) of the result returned in name
25 26	int MPI_	_Get_processor_name(char *	name, int *resultlen)
27 28 29 30	CHAR INTE	processor_name(name, resu ACTER(LEN=MPI_MAX_PROCESS GER, INTENT(OUT) :: resu GER, OPTIONAL, INTENT(OUT	OR_NAME), INTENT(OUT) :: name ltlen
31 32 33 34	CHAR	PROCESSOR_NAME( NAME, RES ACTER*(*) NAME CGER RESULTLEN,IERROR	ULTLEN, IERROR)
35 36 37 38 39 40 41 42 43 44 45 46	of the ca must be p 9 in rack running h MPI_MAX up to this The : In C, a m be larger	II. The name is a character spossible to identify a specific per- 4 of mpp.cs.org" and "231" nomogeneous system). The arg 2_PROCESSOR_NAME character is many characters into name. number of characters actually we ill character is additionally sto than MPI_MAX_PROCESSOR_N	he processor on which it was called at the moment string for maximum flexibility. From this value it iece of hardware; possible values include "processor (where 231 is the actual processor number in the ument name must represent storage that is at least s long. MPI_GET_PROCESSOR_NAME may write written is returned in the output argument, resultlen. red at name[resultlen]. The value of resultlen cannot AME-1. In Fortran, name is padded on the right with cannot be larger than MPI_MAX_PROCESSOR_NAME.

<sup>47</sup> Rationale. This function allows MPI implementations that do process migration to
 <sup>48</sup> 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.*)

## 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 some RMA functionality as defined in Section 11.5.3.

MPI\_ALLOC\_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative inte-	21
	5120	ger)	22
		- · · ·	23
IN	info	info argument (handle)	24
OUT	baseptr	pointer to beginning of memory segment allocated	25
			26
int MPI_A	lloc_mem(MPI_Aint size, N	<pre>/PI_Info info, void *baseptr)</pre>	27
		-	28
	_mem(size, info, baseptr		29
	INTRINSIC :: ISO_C_BIND	· –	30
	ER(KIND=MPI_ADDRESS_KIND)	-	31
	<pre>MPI_Info), INTENT(IN) ::</pre>		32
	C_PTR), INTENT(OUT) :: 1	1	33
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	34
MPI ALLOC	_MEM(SIZE, INFO, BASEPTR	IERROR)	35
	ER INFO, IERROR		36
	ER(KIND=MPI_ADDRESS_KIND)	SIZE, BASEPTR	37
			38
		<b>YPE(C_PTR)</b> , then the following interface must be	39
-	-	d be provided in the deprecated mpif.h through	$_{40}$ ticketWG.
	, ,	e name as the routine with	41
INTEGER(K	IND=MPI_ADDRESS_KIND) BAS	SEPTR, but with a different linker name:	42
			43
	MPI_ALLOC_MEM		44
		IZE, INFO, BASEPTR, IERROR)	45
-	INTRINSIC :: ISO_C_BINDIN	NG, UNLY : C_PTR	46
	ER :: INFO, IERROR	0105	47
INTEG	ER(KIND=MPI_ADDRESS_KIND)	) :: SIZE	48

 $\mathbf{5}$ 

1	TYPE(C_PTR) :: BASEPTR
2	END SUBROUTINE
3	END INTERFACE
4 5	The linker name base of this overloaded function is MPI_ALLOC_MEM_CPTR. The
6	implied linker names are described in Section 17.1.5 on page 609.
7	The info argument can be used to provide directives that control the desired location
8	of the allocated memory. Such a directive does not affect the semantics of the call. Valid
9	info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL
10	is always valid.
11	The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
12	to indicate it failed because memory is exhausted.
13	
14	MPI_FREE_MEM(base)
15	
16	IN base initial address of memory segment allocated by
17 18	MPI_ALLOC_MEM (choice)
19	
20	<pre>int MPI_Free_mem(void *base)</pre>
21	MPI_Free_mem(base, ierror) BIND(C)
22	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: base
23	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24	MPI_FREE_MEM(BASE, IERROR)
25	<type> BASE(*)</type>
26	INTEGER IERROR
27	The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to
28	indicate an invalid base argument.
29 30	
31	Rationale. The C bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar
32	to the bindings for the malloc and free C library calls: a call to
33	MPI_Alloc_mem(, &base) should be paired with a call to MPI_Free_mem(base) (one
34	less level of indirection). Both arguments are declared to be of same type
35	void* so as to facilitate type casting. The Fortran binding is consistent with the C
36	bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the TYPE(C_PTR)
37	pointer or the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable
38	stored at that location. ( <i>End of rationale.</i> )
39	stored at that location. ( <i>Dia of fationale.</i> )
40 41	Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a
41	design similar to the design of C malloc and free functions has to be used, in order
43	to find out the size of a memory segment, when the segment is freed. If no special
44	memory is used, $MPI\_ALLOC\_MEM$ simply invokes malloc, and $MPI\_FREE\_MEM$
45	invokes free.
46	A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate mem-
47	ory in a shared memory segment. (End of advice to implementors.)
48	

Example of use of MPI\_ALLOC\_MEM, in Fortran with Example 8.1 TYPE(C\_PTR) pointers. We assume 4-byte REALs. ! or USE mpi (not guaranteed with INCLUDE 'mpif.h') USE mpi\_f08 USE, INTRINSIC :: ISO\_C\_BINDING TYPE(C\_PTR) :: p REAL, DIMENSION(:,:), POINTER :: a ! no memory is allocated INTEGER, DIMENSION(2) :: shape INTEGER(KIND=MPI\\_ADDRESS\\_KIND) :: size shape = (/100, 100/)size = 4 \* shape(1) \* shape(2)! assuming 4 bytes per REAL CALL MPI\_Alloc\_mem(size, MPI\_INFO\_NULL, p, ierr) ! memory is allocated and CALL C\_F\_POINTER(p, a, shape) ! intrinsic ! now accessible via a(i,j) ! in ISO\_C\_BINDING . . . a(3,5) = 2.71;. . . CALL MPI\_Free\_mem(a, ierr) ! memory is freed

**Example 8.2** Example of use of MPI\_ALLOC\_MEM, in Fortran with non-standard *Craypointers*. We assume 4-byte REALs, and assume that these pointers are address-sized.

```
REAL A
POINTER (P, A(100,100)) ! no memory is allocated
INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE
SIZE = 4*100*100
CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.

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 viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to implementors.)

Example 8.3 Same example, in C.

```
float (* f)[100][100];
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

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

#### 8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during 3 MPI calls. These can include errors that generate exceptions or traps, such as floating point 4 errors or access violations. The set of errors that are handled by MPI is implementationdependent. Each such error generates an MPI exception. 6

The above text takes precedence over any text on error handling within this document. 7 Specifically, text that states that errors will be handled should be read as may be handled. 8

A user can associate error handlers to three types of objects: communicators, windows, 9 and files. The specified error handling routine will be used for any MPI exception that occurs 10 during a call to MPI for the respective object. MPI calls that are not related to any objects 11 are considered to be attached to the communicator MPI\_COMM\_WORLD. The attachment 12of error handlers to objects is purely local: different processes may attach different error 13 handlers to corresponding objects. 14

1516

17

18

1920

21

22

1 2

5

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 MPLABORT 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 23can code their own error handlers.  $^{24}$ 

The error handler MPI\_ERRORS\_ARE\_FATAL is associated by default with MPI\_COMM-25\_WORLD after initialization. Thus, if the user chooses not to control error handling, every 26error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, 27a user may choose to handle errors in its main code, by testing the return code of MPI 28calls and executing a suitable recovery code when the call was not successful. In this case, 29 the error handler MPI\_ERRORS\_RETURN will be used. Usually it is more convenient and 30 more efficient not to test for errors after each MPI call, and have such error handled by a  $^{31}$ non-trivial MPI error handler. 32

After an error is detected, the state of MPI is undefined. That is, using a user-defined 33 error handler, or MPI\_ERRORS\_RETURN, does not necessarily allow the user to continue to 34use MPI after an error is detected. The purpose of these error handlers is to allow a user to 35 issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O36 buffers) before a program exits. An MPI implementation is free to allow MPI to continue 37 after an error but is not required to do so. 38

39 40

41

42

43

Advice to implementors. A high-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.)

4445

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are 46 provided to create new error handlers, to associate error handlers with objects, and to test 47which error handler is associated with an object. C has distinct typedefs for user defined 48

$\operatorname{error}$	handling callback functions that ac	ccept communicator, file, and window arguments.	1
In Fo	rtran there are three user routines.		2
A	An error handler object is created	by a call to MPI_XXX_CREATE_ERRHANDLER,	3
	e XXX is, respectively, COMM, WIN,	,	4
	, , , , , ,	ommunicator, window, or file by a call to	5
		or handler must be either a predefined error han-	6
		d by a call to MPI_XXX_CREATE_ERRHANDLER,	7
		or handlers MPI_ERRORS_RETURN and	8
		d to communicators, windows, and files.	9
			10
	•	ted with a communicator, window, or file can be	11
	ved by a call to MPI_XXX_GET_EF		12
		ER_FREE can be used to free an error handler that	
	reated by a call to MPI_XXX_CREA		13
		HANDLER behave as if a new error handler ob-	14
ē	s created. That is, once the erro	e ,	15
MPI_	$ERRHANDLER_FREE$ should be cal	lled with the error handler returned from	16
MPI_	{COMM,WIN,FILE}_GET_ERRHAN	DLER to mark the error handler for deallocation.	17
This j	provides behavior similar to that of	MPI_COMM_GROUP and MPI_GROUP_FREE.	18
			19
		ality implementations should raise an error when	20
		by a call to MPI_XXX_CREATE_ERRHANDLER is	21
	•	ype with a call to MPI_YYY_SET_ERRHANDLER.	22
	To do so, it is necessary to maint	ain, with each error handler, information on the	23
	typedef of the associated user funct	tion. (End of advice to implementors.)	24
-		1	25
Г	The syntax for these calls is given b	elow.	25 26
			26
Г 8.3.1	The syntax for these calls is given b Error Handlers for Communicato		26 27
			26 27 28
			26 27 28 29
8.3.1	Error Handlers for Communicato	rs	26 27 28 29 30
8.3.1 MPI_0	Error Handlers for Communicato	rs omm_errhandler_fn, errhandler)	26 27 28 29
8.3.1 MPI_0 IN	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function)	26 27 28 29 30 31 32
8.3.1 MPI_0	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn	rs omm_errhandler_fn, errhandler)	26 27 28 29 30 31 32 33
8.3.1 MPI_0 IN	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function)	26 27 28 29 30 31 32 33 34
8.3.1 MPI_ IN OU <sup>-</sup>	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle)	26 27 28 30 31 32 33 34 35
8.3.1 MPI_ IN OU <sup>-</sup>	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler 1PI_Comm_create_errhandler(MPI	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle)	26 27 28 29 30 31 32 33 34 35 36
8.3.1 MPI_ IN OU <sup>-</sup> int M	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn,	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler)	26 27 28 29 30 31 32 33 34 35 36 37
8.3.1 MPI_C IN OU <sup>-</sup> int M MPI_C	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C)	26 27 28 29 30 31 32 33 34 35 36 37 38
8.3.1 MPI_ IN OU <sup>-</sup> int M MPI_C	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn	26 27 28 29 30 31 32 33 34 35 36 37 38 39
8.3.1 MPI_ IN OU <sup>-</sup> int M MPI_C F	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_ CYPE(MPI_Errhandler), INTENT(0)	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler	26 27 28 29 30 31 32 33 34 35 36 37 38
8.3.1 MPI_ IN OU <sup>-</sup> int M MPI_C F	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler	26 27 28 29 30 31 32 33 34 35 36 37 38 39
8.3.1 MPI_U IN OU <sup>-</sup> int M MPI_C F I	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_ CYPE(MPI_Errhandler), INTENT(OUT)	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler :: ierror	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
8.3.1 MPI_C IN OU <sup>-</sup> int M MPI_C MPI_C	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_err PROCEDURE(MPI_Comm_errhandler_ CYPE(MPI_Errhandler), INTENT(OUT) INTEGER, OPTIONAL, INTENT(OUT) COMM_CREATE_ERRHANDLER(COMM_ER	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
8.3.1 MPI_ IN OU Int M MPI_C F I MPI_C E	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_ TYPE(MPI_Errhandler), INTENT(OUT) INTEGER, OPTIONAL, INTENT(OUT) COMM_CREATE_ERRHANDLER(COMM_ER EXTERNAL COMM_ERRHANDLER_FN	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler :: ierror	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
8.3.1 MPI_ IN OU Int M MPI_C F I MPI_C E	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_err PROCEDURE(MPI_Comm_errhandler_ CYPE(MPI_Errhandler), INTENT(OUT) INTEGER, OPTIONAL, INTENT(OUT) COMM_CREATE_ERRHANDLER(COMM_ER	rs pmm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler :: ierror	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
8.3.1 MPI_0 IN OU <sup>-1</sup> int M MPI_C F I MPI_C E I	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_ TYPE(MPI_Errhandler), INTENT(OUT) INTEGER, OPTIONAL, INTENT(OUT) COMM_CREATE_ERRHANDLER(COMM_ER EXTERNAL COMM_ERRHANDLER_FN	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler :: ierror RHANDLER_FN, ERRHANDLER, IERROR)	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
8.3.1 MPI_U IN OU Int M MPI_C F I MPI_C E I I C	Error Handlers for Communicato COMM_CREATE_ERRHANDLER(co comm_errhandler_fn T errhandler MPI_Comm_create_errhandler(MPI *comm_errhandler_fn, Comm_create_errhandler(comm_er PROCEDURE(MPI_Comm_errhandler_fn, COMM_CREATE_ERRHANDLER, INTENT(OUT) COMM_CREATE_ERRHANDLER, IERROR EXTERNAL COMM_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR Creates an error handler that can be	rs omm_errhandler_fn, errhandler) user defined error handling procedure (function) MPI error handler (handle) _Comm_errhandler_function MPI_Errhandler *errhandler) rhandler_fn, errhandler, ierror) BIND(C) function) :: comm_errhandler_fn UT) :: errhandler :: ierror RHANDLER_FN, ERRHANDLER, IERROR)	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

1 typedef void MPI\_Comm\_errhandler\_function(MPI\_Comm \*, int \*, ...);  $\mathbf{2}$ The first argument is the communicator in use. The second is the error code to be 3 returned by the MPI routine that raised the error. If the routine would have returned 4 MPI\_ERR\_IN\_STATUS, it is the error code returned in the status for the request that caused 5the error handler to be invoked. The remaining arguments are "varargs" arguments whose 6 number and meaning is implementation-dependent. An implementation should clearly doc-7 ument these arguments. Addresses are used so that the handler may be written in Fortran. 8 With the Fortran mpi\_f08 module, the user routine comm\_errhandler\_fn should be of the 9 form: 10 ABSTRACT INTERFACE 11 SUBROUTINE MPI\_Comm\_errhandler\_function(comm, error\_code) BIND(C) 12TYPE(MPI\_Comm) :: comm 13 INTEGER :: error\_code 14ticket WG.  $^{\rm 15}$ With the Fortran mpi module and the deprecated mpif.h, the user routine 16COMM\_ERRHANDLER\_FN should be of the form: 17 SUBROUTINE COMM\_ERRHANDLER\_FUNCTION(COMM, ERROR\_CODE) 18 INTEGER COMM, ERROR\_CODE 19 20Rationale. The variable argument list is provided because it provides an ISO-21standard hook for providing additional information to the error handler; without this 22hook, ISO C prohibits additional arguments. (*End of rationale.*) 23 $^{24}$ A newly created communicator inherits the error handler that Advice to users. 25is associated with the "parent" communicator. In particular, the user can specify 26a "global" error handler for all communicators by associating this handler with the 27communicator MPI\_COMM\_WORLD immediately after initialization. (End of advice to 28users.) 2930 31MPI\_COMM\_SET\_ERRHANDLER(comm, errhandler) 32 33 INOUT comm communicator (handle) 34 IN errhandler new error handler for communicator (handle) 35 36 int MPI\_Comm\_set\_errhandler(MPI\_Comm comm, MPI\_Errhandler errhandler) 37 38 MPI\_Comm\_set\_errhandler(comm, errhandler, ierror) BIND(C) 39 TYPE(MPI\_Comm), INTENT(IN) :: comm 40 TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI\_COMM\_SET\_ERRHANDLER(COMM, ERRHANDLER, IERROR) 43 INTEGER COMM, ERRHANDLER, IERROR 4445Attaches a new error handler to a communicator. The error handler must be either 46a predefined error handler, or an error handler created by a call to 47 MPI\_COMM\_CREATE\_ERRHANDLER. 48

MPI_CON	/M_GET_ERRHANDLER(co	mm, errhandler)	1
IN	comm	communicator (handle)	2
OUT	errhandler	error handler currently associated with communicator	3
001	ermanuler	(handle)	4
		(nandre)	5 6
int MPI_	Comm_get_errhandler(MPI	_Comm comm, MPI_Errhandler *errhandler)	7
MDT Comm	act orrhordlor(comm o	rrhandler, ierror) BIND(C)	8
	(MPI_Comm), INTENT(IN)		9
	(MPI_Errhandler), INTEN		10
	GER, OPTIONAL, INTENT(O		11 12
МРТ СОММ	_GET_ERRHANDLER(COMM, E	RRHANDI FR TFRROR)	12
	GER COMM, ERRHANDLER, I		14
			15
		ently associated with a communicator. The analysis at its entry point the current error handler	16
	<b>2</b> /	ate error handler for this communicator, and restore	17
	ting the previous error hand	,	18
	5 I		19 20
8.3.2 Er	ror Handlers for Windows		20
			22
			23
MPI_WIN	_CREATE_ERRHANDLER(v	vin_errhandler_fn, errhandler)	24
IN	win_errhandler_fn	user defined error handling procedure (function)	25
OUT	errhandler	MPI error handler (handle)	26 27
			21
int MPI_	Win_create_errhandler(M	PI_Win_errhandler_function	29
	<pre>*win_errhandler_fr</pre>	n, MPI_Errhandler *errhandler)	30
MPT Win	create errhandler(win e	rrhandler_fn, errhandler, ierror) BIND(C)	31
		r_function) :: win_errhandler_fn	32
	(MPI_Errhandler), INTEN		33
INTE	GER, OPTIONAL, INTENT(O	UT) :: ierror	$\frac{34}{35}$
MPI WIN	CREATE_ERRHANDLER(WIN F	RRHANDLER_FN, ERRHANDLER, IERROR)	36
	RNAL WIN_ERRHANDLER_FN		37
INTE	GER ERRHANDLER, IERROR		38
Crea	tes an error handler that ca	in be attached to a window object. The user routine	39
		PI_Win_errhandler_function which is defined as	40
		_function(MPI_Win *, int *,);	41
		v in use, the second is the error code to be returned.	42 43
	0	e user routine win_errhandler_fn should be of the form:	43 44
	INTERFACE		45
		_function(win, error_code) BIND(C)	46
TY	PE(MPI_Win) :: win		47
T 11			10

INTEGER :: error\_code

ticketWG. 1 2 3 4 5	WIN_ERR SUBROUTI	RHANDLER_FN s	should be of the for DLER_FUNCTION(W	precated mpif.h, the user routine orm: IN, ERROR_CODE)	
6 7					
8	MPI_WIN	I_SET_ERRHANI	DLER(win, errhand	ller)	
9	INOUT	win	wi	indow (handle)	
10	IN	errhandler	ne	ew error handler for window (handle)	
11					
12 13	int MPI_	Win_set_errhar	ndler(MPI_Win w	in, MPI_Errhandler errhandler)	
14	MPI_Win_	set_errhandler	(win, errhandle	er, ierror) BIND(C)	
15			<pre>TENT(IN) :: wii </pre>		
16			er), INTENT(IN), INTENT(OUT) :	:: errhandler	
17					
18 19			R(WIN, ERRHANDL)	ER, IERROR)	
20	INTE	GER WIN, ERRHA	ANDLER, IERRUR		
21				ndow. The error handler must be either a pre-	
22		,		r created by a call to	
23		I_CREATE_ERRH	IANDLEK.		
24 25					
26	MPI_WIN	I_GET_ERRHAN	DLER(win, errhand	dler)	
27	IN	win	wi	indow (handle)	
28	OUT	errhandler	er	ror handler currently associated with window (han-	
29			dl	e)	
30 31					
32	int MPI_	Win_get_errhar	ndler(MPI_Win w:	in, MPI_Errhandler *errhandler)	
33	MPI_Win_	get_errhandler	(win, errhandle	er, ierror) BIND(C)	
34	TYPE(MPI_Win), INTENT(IN) :: win				
35			er), INTENT(OUT) , INTENT(OUT) :		
36 37					
38			R(WIN, ERRHANDL	ER, IERROR)	
39	LNIE	GER WIN, ERRHA	ANDLER, IERRUR		
40	Retri	ieves the error ha	andler currently as	ssociated with a window.	
41 42					
42					
44					
45					
46					
47					
48					

INTEGER FILE, ERRHANDLER, IERROR

#### 8.3.3 Error Handlers for Files 1 2 MPI\_FILE\_CREATE\_ERRHANDLER(file\_errhandler\_fn, errhandler) IN file\_errhandler\_fn user defined error handling procedure (function) 6 OUT errhandler MPI error handler (handle) int MPI\_File\_create\_errhandler(MPI\_File\_errhandler\_function 10 \*file\_errhandler\_fn, MPI\_Errhandler \*errhandler) 11 MPI\_File\_create\_errhandler(file\_errhandler\_fn, errhandler, ierror) BIND(C) 12 PROCEDURE(MPI\_File\_errhandler\_function) :: file\_errhandler\_fn 13 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 14 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI\_FILE\_CREATE\_ERRHANDLER(FILE\_ERRHANDLER\_FN, ERRHANDLER, IERROR) 17EXTERNAL FILE\_ERRHANDLER\_FN 18 INTEGER ERRHANDLER, IERROR 19 Creates an error handler that can be attached to a file object. The user routine should 20be, in C, a function of type MPI\_File\_errhandler\_function, which is defined as 21typedef void MPI\_File\_errhandler\_function(MPI\_File \*, int \*, ...); 22 23The first argument is the file in use, the second is the error code to be returned. $^{24}$ With the Fortran mpi\_f08 module, the user routine file\_errhandler\_fn should be of the form: 25ABSTRACT INTERFACE 26SUBROUTINE MPI\_File\_errhandler\_function(file, error\_code) BIND(C) 27TYPE(MPI\_File) :: file 28INTEGER :: error\_code 29 With the Fortran mpi module and the deprecated mpif.h, the user routine 30 ticketWG. FILE\_ERRHANDLER\_FN should be of the form: 31 SUBROUTINE FILE\_ERRHANDLER\_FUNCTION(FILE, ERROR\_CODE) 32 INTEGER FILE, ERROR\_CODE 33 34 35MPI\_FILE\_SET\_ERRHANDLER(file, errhandler) 36 37 INOUT file file (handle) 38 IN errhandler new error handler for file (handle) 39 40 int MPI\_File\_set\_errhandler(MPI\_File file, MPI\_Errhandler errhandler) 41 42MPI\_File\_set\_errhandler(file, errhandler, ierror) BIND(C) 43 TYPE(MPI\_File), INTENT(IN) :: file 44 TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46MPI\_FILE\_SET\_ERRHANDLER(FILE, ERRHANDLER, IERROR) 47

Attaches a new error handler to a file. The error handler must be either a predefined

CH.

 $\mathbf{2}$ error handler, or an error handler created by a call to MPI\_FILE\_CREATE\_ERRHANDLER. 3 4 MPI\_FILE\_GET\_ERRHANDLER(file, errhandler) 56 IN file file (handle) 7 OUT errhandler error handler currently associated with file (handle) 8 9 int MPI\_File\_get\_errhandler(MPI\_File file, MPI\_Errhandler \*errhandler) 10 11MPI\_File\_get\_errhandler(file, errhandler, ierror) BIND(C) 12TYPE(MPI\_File), INTENT(IN) :: file 13 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15MPI\_FILE\_GET\_ERRHANDLER(FILE, ERRHANDLER, IERROR) 16INTEGER FILE, ERRHANDLER, IERROR 1718 Retrieves the error handler currently associated with a file. 19208.3.4 Freeing Errorhandlers and Retrieving Error Strings 2122 23MPI\_ERRHANDLER\_FREE( errhandler ) 24INOUT errhandler MPI error handler (handle) 252627int MPI\_Errhandler\_free(MPI\_Errhandler \*errhandler) 28MPI\_Errhandler\_free(errhandler, ierror) BIND(C) 29 TYPE(MPI\_Errhandler), INTENT(INOUT) :: errhandler 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror  $^{31}$ 32 MPI\_ERRHANDLER\_FREE(ERRHANDLER, IERROR) 33 INTEGER ERRHANDLER, IERROR 34 Marks the error handler associated with errhandler for deallocation and sets errhandler 35 to MPI\_ERRHANDLER\_NULL. The error handler will be deallocated after all the objects 36 associated with it (communicator, window, or file) have been deallocated. 37 38 39 MPI\_ERROR\_STRING( errorcode, string, resultlen ) 40IN errorcode Error code returned by an MPI routine 41 42OUT string Text that corresponds to the errorcode 43 OUT resultlen Length (in printable characters) of the result returned 44 in string 4546int MPI\_Error\_string(int errorcode, char \*string, int \*resultlen) 4748MPI\_Error\_string(errorcode, string, resultlen, ierror) BIND(C)

```
1
    INTEGER, INTENT(IN) ::
                                errorcode
                                                                                             \mathbf{2}
    CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
                                                                                             3
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                                                                             4
                                             ierror
                                                                                             5
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
                                                                                             6
    INTEGER ERRORCODE, RESULTLEN, IERROR
                                                                                             7
    CHARACTER*(*) STRING
                                                                                             8
                                                                                             9
    Returns the error string associated with an error code or class. The argument string
                                                                                            10
must represent storage that is at least MPI_MAX_ERROR_STRING characters long.
                                                                                            11
    The number of characters actually written is returned in the output argument, resultlen.
                                                                                            12
     Rationale. The form of this function was chosen to make the Fortran and C bindings
                                                                                            13
     similar. A version that returns a pointer to a string has two difficulties. First, the
                                                                                            14
     return string must be statically allocated and different for each error message (allowing
                                                                                            15
     the pointers returned by successive calls to MPI_ERROR_STRING to point to the
                                                                                            16
     correct message). Second, in Fortran, a function declared as returning CHARACTER*(*)
                                                                                            17
     can not be referenced in, for example, a PRINT statement. (End of rationale.)
                                                                                            18
                                                                                            19
                                                                                            20
      Error Codes and Classes
8.4
                                                                                            21
                                                                                            22
The error codes returned by MPI are left entirely to the implementation (with the exception
                                                                                            23
of MPI_SUCCESS). This is done to allow an implementation to provide as much information
                                                                                            ^{24}
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. 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
OUT	errorclass	Error class associated with errorcode

int MPI\_Error\_class(int errorcode, int \*errorclass)

25

26

27

28

29

30

31

32

33

34 35

36

37

38

39

_		
1		
2	MPI_SUCCESS	No error
3	MPI_ERR_BUFFER	Invalid buffer pointer
4		-
5	MPI_ERR_COUNT	Invalid count argument
6	MPI_ERR_TYPE	Invalid datatype argument
7	MPI_ERR_TAG	Invalid tag argument
8	MPI_ERR_COMM	Invalid communicator
9	MPI_ERR_RANK	Invalid rank
10	MPI_ERR_REQUEST	Invalid request (handle)
11	MPI_ERR_ROOT	Invalid root
12	MPI_ERR_GROUP	Invalid group
13	MPI_ERR_OP	Invalid operation
14	MPI_ERR_TOPOLOGY	Invalid topology
	MPI_ERR_DIMS	Invalid dimension argument
15	MPI_ERR_ARG	Invalid argument of some other kind
16	MPI_ERR_UNKNOWN	Unknown error
17	MPI_ERR_TRUNCATE	Message truncated on receive
18	MPI_ERR_OTHER	Known error not in this list
19	MPI_ERR_INTERN	Internal MPI (implementation) error
20	MPI_ERR_IN_STATUS	Error code is in status
21	MPI_ERR_PENDING	Pending request
22	MPI_ERR_KEYVAL	Invalid keyval has been passed
23		
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
36		MPI_LOOKUP_NAME
37	MPI_ERR_WIN	Invalid win argument
38	MPI_ERR_SIZE	Invalid size argument
39	MPI_ERR_DISP	Invalid disp argument
40	MPI_ERR_INFO	Invalid info argument
	MPI_ERR_LOCKTYPE	Invalid locktype argument
41	MPI_ERR_ASSERT	Invalid assert argument
42	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
43	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
44		, tong by non-on-boundarion of further owns
45		
46	Table 8.	.1: Error classes (Part 1)
47		

MPI_ERR_RMA_RANGE	Target memory is not part of the win-	4
	dow (in the case of a window created	5
	with MPI_WIN_CREATE_DYNAMIC, tar-	6
	get memory is not attached)	7
MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because	8
		9
	of resource exhaustion)	10
MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro- cess in the group of the specified commu-	11
	nicator cannot expose shared memory)	12
		13
MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the called function	14
MPI_ERR_FILE	Invalid file handle	15
MPI_ERR_NOT_SAME	Collective argument not identical on all	16
WFI_ERK_NOT_SAME	processes, or collective routines called in	17
	a different order by different processes	18
MPI_ERR_AMODE	Error related to the amode passed to	19
WFI_ERK_AWODE	MPI_FILE_OPEN	20
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	21
WFI_ERR_ONSOFFORTED_DATAREF	MPI_FILE_SET_VIEW	22
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	23
	a file which supports sequential access only	24
MPI_ERR_NO_SUCH_FILE	File does not exist	25
MPI_ERR_FILE_EXISTS	File exists	26
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	27
MPI_ERR_ACCESS	Permission denied	28
MPI_ERR_NO_SPACE	Not enough space	29
MPI_ERR_QUOTA	Quota exceeded	30 31
MPI_ERR_READ_ONLY	Read-only file or file system	31
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	33
	the file is currently open by some process	34
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	35
	tered because a data representation identi-	36
	fier that was already defined was passed to	37
	MPI_REGISTER_DATAREP	38
MPI_ERR_CONVERSION	An error occurred in a user supplied data	39
	conversion function.	40
MPI_ERR_IO	Other I/O error	41
MPI_ERR_LASTCODE	Last error code	42
		43
Table 8 9. Fra	cor classes (Part 2)	44
Tuble 0.2. L11		45
		46

1 MPI\_Error\_class(errorcode, errorclass, ierror) BIND(C)  $\mathbf{2}$ INTEGER, INTENT(IN) :: errorcode 3 INTEGER, INTENT(OUT) :: errorclass 4 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5MPI\_ERROR\_CLASS(ERRORCODE, ERRORCLASS, IERROR) 6 INTEGER ERRORCODE, ERRORCLASS, IERROR 7 8 The function MPI\_ERROR\_CLASS maps each standard error code (error class) onto 9 itself. 10 11 8.5 Error Classes, Error Codes, and Error Handlers 1213 Users may want to write a layered library on top of an existing MPI implementation, and 14this library may have its own set of error codes and classes. An example of such a library 15is an I/O library based on MPI, see Chapter 13 on page 489. For this purpose, functions 16are needed to: 1718 1. add a new error class to the ones an MPI implementation already knows. 19 2. associate error codes with this error class, so that MPI\_ERROR\_CLASS works. 20213. associate strings with these error codes, so that MPI\_ERROR\_STRING works. 22 234. invoke the error handler associated with a communicator, window, or object. 24Several functions are provided to do this. They are all local. No functions are provided 25to free error classes or codes: it is not expected that an application will generate them in 26significant numbers. 272829MPI\_ADD\_ERROR\_CLASS(errorclass) 30 OUT errorclass value for the new error class (integer)  $^{31}$ 32 33 int MPI\_Add\_error\_class(int \*errorclass) 34MPI\_Add\_error\_class(errorclass, ierror) BIND(C) 35 INTEGER, INTENT(OUT) :: errorclass 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 38 MPI\_ADD\_ERROR\_CLASS(ERRORCLASS, IERROR) 39 INTEGER ERRORCLASS, IERROR 40 Creates a new error class and returns the value for it. 41 42Rationale. To avoid conflicts with existing error codes and classes, the value is set 43 by the implementation and not by the user. (End of rationale.) 44 45Advice to implementors. A high-quality implementation will return the value for 46 a new errorclass in the same deterministic way on all processes. (End of advice to 47 *implementors.*)

Advice to users. Since a call to MPI\_ADD\_ERROR\_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. However, if an implementation returns the new errorclass in a deterministic way, and they are always generated in the same order on the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice to users.*)

The value of MPI\_ERR\_LASTCODE is a constant value and is not affected by new userdefined error codes and classes. Instead, a predefined attribute key MPI\_LASTUSEDCODE is associated with MPI\_COMM\_WORLD. The attribute value corresponding to this key is the current 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 equal to MPI\_ERR\_LASTCODE.

Advice to users. The value returned by the key MPI\_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multithreaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI\_LASTUSEDCODE is valid. (*End of advice to users.*)

#### MPI\_ADD\_ERROR\_CODE(errorclass, errorcode) errorclass IN error class (integer) OUT errorcode new error code to associated with errorclass (integer) int MPI\_Add\_error\_code(int errorclass, int \*errorcode) MPI\_Add\_error\_code(errorclass, errorcode, ierror) BIND(C) INTEGER, INTENT(IN) :: errorclass INTEGER, INTENT(OUT) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_ADD\_ERROR\_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR

Creates new error code associated with errorclass and returns its value in errorcode.

*Rationale.* To avoid conflicts with existing error codes and classes, the value of the new error code is set by the implementation and not by the user. (*End of rationale.*)

Advice to implementors. A high-quality implementation will return the value for a new errorcode in the same deterministic way on all processes. (End of advice to implementors.)

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

```
1
     MPI_ADD_ERROR_STRING(errorcode, string)
2
       IN
                 errorcode
                                              error code or class (integer)
3
       IN
                 string
                                              text corresponding to errorcode (string)
4
5
6
     int MPI_Add_error_string(int errorcode, const char *string)
\overline{7}
     MPI_Add_error_string(errorcode, string, ierror) BIND(C)
8
          INTEGER, INTENT(IN) :: errorcode
9
          CHARACTER(LEN=*), INTENT(IN) :: string
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
11
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
12
          INTEGER ERRORCODE, IERROR
13
14
          CHARACTER*(*) STRING
15
          Associates an error string with an error code or class. The string must be no more
16
     than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the
17
     calling language. The length of the string does not include the null terminator in C. Trailing
18
     blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that
19
     already has a string will replace the old string with the new string. It is erroneous to call
20
     MPI_ADD_ERROR_STRING for an error code or class with a value \leq MPI_ERR_LASTCODE.
21
          If MPI_ERROR_STRING is called when no string has been set, it will return a empty
22
     string (all spaces in Fortran, "" in C).
23
          Section 8.3 on page 340 describes the methods for creating and associating error han-
^{24}
     dlers with communicators, files, and windows.
25
26
27
     MPI_COMM_CALL_ERRHANDLER (comm, errorcode)
28
       IN
                 comm
                                              communicator with error handler (handle)
29
       IN
                 errorcode
                                              error code (integer)
30
^{31}
32
     int MPI_Comm_call_errhandler(MPI_Comm_comm, int errorcode)
33
     MPI_Comm_call_errhandler(comm, errorcode, ierror) BIND(C)
34
          TYPE(MPI_Comm), INTENT(IN) ::
                                              comm
35
          INTEGER, INTENT(IN) :: errorcode
36
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
37
38
     MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
39
          INTEGER COMM, ERRORCODE, IERROR
40
          This function invokes the error handler assigned to the communicator with the error
41
     code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if
42
     the error handler was successfully called (assuming the process is not aborted and the error
43
     handler returns).
44
45
                              Users should note that the default error handler is
           Advice to users.
46
           MPI_ERRORS_ARE_FATAL. Thus, calling MPI_COMM_CALL_ERRHANDLER will abort
47
48
```

the comm processes if the default error handler has not been changed for this communicator or on the parent before the communicator was created. (*End of advice to users.*)

			5			
MPI_WIN_CALL_ERRHANDLER (win, errorcode)						
IN	win	window with error handler (handle)	7 8			
			9			
IN	errorcode	error code (integer)	10			
			11			
int MPI	<pre>int MPI_Win_call_errhandler(MPI_Win win, int errorcode)</pre>					
MPI_Wir	n_call_errhandler(	win, errorcode, ierror) BIND(C)	13			
TYF	PE(MPI_Win), INTEN	Γ(IN) :: win	14			
	TEGER, INTENT(IN)		15			
INT	TEGER, OPTIONAL, I	NTENT(OUT) :: ierror	16 17			
MPI_WIN	MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)					
	INTEGER WIN, ERRORCODE, IERROR					
Th:	a function involves th	a amon handlen aggirmed to the window with the owner and	19 20			
		e error handler assigned to the window with the error code rns MPI_SUCCESS in C and the same value in IERROR if the	21			
* *		y called (assuming the process is not aborted and the error	22			
	returns).	y caned (assuming the process is not aborted and the error	23			
			24			
Aa	dvice to users. As w	with communicators, the default error handler for windows is	25			
M	PI_ERRORS_ARE_FATA	AL. (End of advice to users.)	26			
			27			
			28 29			
MPI_FIL	MPI_FILE_CALL_ERRHANDLER (fh, errorcode)					
IN	fh	file with error handler (handle)	30 31			
			32			
IN	errorcode	error code (integer)	33			
			34			
int MPI	_File_call_errhan	dler(MPI_File fh, int errorcode)	35			
MPI_Fil	le_call_errhandler	(fh, errorcode, ierror) BIND(C)	36			
TYF	PE(MPI_File), INTE	NT(IN) :: fh	37			
	TEGER, INTENT(IN)		38			
INT	TEGER, OPTIONAL, I	NTENT(OUT) :: ierror	39			
MPI_FII	MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)					
	TEGER FH, ERRORCOD		41			
		error handler assigned to the file with the error code supplied.	42 43			
			43 44			
This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).						
was succ	contraction (assumed (assumed assumed assumed assumed as a second s	ming the process is not aborted and the error handler returns).	45 46			

Advice to users. Unlike errors on communicators and windows, the default behavior for files is to have MPI\_ERRORS\_RETURN. (End of advice to users.)

Advice to users. Users are warned that handlers should not be called recursively with MPI\_COMM\_CALL\_ERRHANDLER, MPI\_FILE\_CALL\_ERRHANDLER, or MPI\_WIN\_CALL\_ERRHANDLER. Doing this can create a situation where an infinite

recursion is created. This can occur if MPI\_COMM\_CALL\_ERRHANDLER,

MPI\_FILE\_CALL\_ERRHANDLER, or MPI\_WIN\_CALL\_ERRHANDLER is called inside an error handler.

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.*)

12 13 14

15

1

2

3

4

5

6

7

8

9

10

11

## 8.6 Timers and Synchronization

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 2.6.4 on page 20.

```
21
22
```

23

25

36

48

```
MPI_WTIME()
```

```
<sup>24</sup> double MPI_Wtime(void)
```

26 DOUBLE PRECISION MPI\_Wtime() BIND(C)

```
27 DOUBLE PRECISION MPI_WTIME()
28
```

MPI\_WTIME returns a floating-point number of seconds, representing elapsed wall clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
37
         double starttime, endtime;
38
         starttime = MPI_Wtime();
39
          .... stuff to be timed
                                     . . .
40
                    = MPI_Wtime();
         endtime
41
         printf("That took %f seconds\n",endtime-starttime);
42
     }
43
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time." (But see also the discussion of MPI\_WTIME\_IS\_GLOBAL in Section 8.1.2).

```
MPI_WTICK()
```

double MPI\_Wtick(void)

DOUBLE PRECISION MPI\_Wtick() BIND(C)

DOUBLE PRECISION MPI\_WTICK()

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
```

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, 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 2	return 0;
2 3 4 5 6 7 8 9 10	<pre>} The Fortran version takes only IERROR. Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C. After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI_INFO_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of mpiexec:</pre>
11 12	command Name of program executed.
12	argv Space separated arguments to command.
14 15	maxprocs Maximum number of MPI processes to start.
16	soft Allowed values for number of processors.
17 18	host Hostname.
19	arch Architecture name.
20 21	wdir Working directory of the MPI process.
22 23	file Value is the name of a file in which additional information is specified.
24 25	thread_level Requested level of thread support, if requested before the program started execution.
26 27 28 29 30 31 32 33 34 35 36	Note that all values are strings. Thus, the maximum number of processes is represented by a string such as "1024" and the requested level is represented by a string such as "MPI_THREAD_SINGLE". The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Imple- mentations may provide additional, implementation specific, (key,value) pairs. In case where the MPI processes were started with MPI_COMM_SPAWN_MULTIPLE or, equivalently, with a startup mechanism that supports multiple process specifications, then the values stored in the info object MPI_INFO_ENV at a process are those values that affect the local MPI process.
37 38	<b>Example 8.4</b> If MPI is started with a call to
39	mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
40 41 42 43	Then the first 5 processes will have have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, sun). The next 10 processes will have in MPI_INFO_ENV (command, atmos), (maxprocs, 10), and (arch, rs600)
44 45 46 47 48	Advice to users. The values passed in MPI_INFO_ENV are the values of the arguments passed to the mechanism that started the MPI execution — not the actual value provided. Thus, the value associated with maxprocs is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. ( <i>End of advice to users.</i> )

### MPI\_FINALIZE()

```
int MPI_Finalize(void)
```

### MPI\_FINALIZE(IERROR) INTEGER IERROR

This routine cleans up all MPI state. If an MPI program terminates normally (i.e., not due to a call to MPI\_ABORT or an unrecoverable error) then each process must call MPI\_FINALIZE before it exits.

Before an MPI process invokes MPI\_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications: It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call MPI\_WAIT, MPI\_TEST, MPI\_REQUEST\_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI\_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI\_XXX\_FREE calls.

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 collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 397.

The following examples illustrates these rules

Example 8.5 The following code is correct

Process O	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Send(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

**Example 8.6** Without a matching receive, the program is erroneous

Process 0	Process 1	44
		45
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>	46
<pre>MPI_Send (dest=1);</pre>		47
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>	48

1

 $\mathbf{2}$ 

3

4 5 6

7

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

1 **Example 8.7** This program is correct: Process 0 calls MPI\_Finalize after it has executed  $\mathbf{2}$ the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call 3 that completes the matching receive operation before it calls MPI\_Finalize. 4

```
Process 0
                                         Proces 1
5
         _____
                                          _____
6
       MPI_Init();
                                         MPI_Init();
7
                                         MPI_Recv(src=0);
       MPI_Isend(dest=1);
8
       MPI_Request_free();
                                         MPI_Finalize();
9
       MPI_Finalize();
                                          exit();
10
     exit();
11
12
13
     Example 8.8 This program is correct. The attached buffer is a resource allocated by the
14
     user, not by MPI; it is available to the user after MPI is finalized.
15
                                          Process 1
        Process 0
16
         _____
                                           _____
17
        MPI_Init();
                                         MPI_Init();
18
        buffer = malloc(1000000);
                                         MPI_Recv(src=0);
19
        MPI_Buffer_attach();
                                         MPI_Finalize();
20
        MPI_Send(dest=1));
                                          exit();
21
        MPI_Finalize();
22
        free(buffer);
23
         exit();
^{24}
25
26
                      This program is correct. The cancel operation must succeed, since the
     Example 8.9
27
     send cannot complete normally. The wait operation, after the call to MPI_Cancel, is local
28
     — no matching MPI call is required on process 1.
29
30
31
        Process 0
                                         Process 1
32
         _____
                                          _____
33
        MPI_Issend(dest=1);
                                         MPI_Finalize();
```

Advice to implementors. Even though a process has executed all MPI calls needed to complete the communications it is involved with, such communication may not yet be completed from the viewpoint of the underlying MPI system. For example, a blocking send may have returned, even though the data is still buffered at the sender in an MPI 42buffer; an MPI process may receive a cancel request for a message it has completed receiving. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI\_FINALIZE returns. Thus, if a process exits after the call to MPI\_FINALIZE, this will not cause an ongoing communication to fail. The MPI implementation should also complete freeing all objects marked for deletion by MPI calls that freed them. (End of advice to implementors.)

34

35

36

37 38

39

40

41

43

44

45

46

47

48

MPI\_Cancel();

MPI\_Finalize();

MPI\_Wait();

Once MPI\_FINALIZE returns, no MPI routine (not even MPI\_INIT) may be called, except for MPI\_GET\_VERSION, MPI\_GET\_LIBRARY\_VERSION, MPI\_INITIALIZED, MPI\_FINALIZED, and any function with the prefix MPI\_T\_ (within the constraints for functions with this prefix listed in Section 14.3.4).

Although it is not required that all processes return from MPI\_FINALIZE, it is required that at least process 0 in MPI\_COMM\_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from MPI\_FINALIZE.

**Example 8.10** 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.

```
. . .
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    . . .
    MPI_Finalize();
    if (myrank == 0) {
        resultfile = fopen("outfile","w");
        dump_results(resultfile);
        fclose(resultfile);
    }
    exit(0);
MPI_INITIALIZED(flag)
                                     Flag is true if MPI_INIT has been called and false
 OUT
          flag
                                     otherwise.
int MPI_Initialized(int *flag)
MPI_Initialized(flag, ierror) BIND(C)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                         ierror
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

This routine may be used to determine whether MPI\_INIT has been called. 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 of the few routines that may be called before MPI\_INIT is called. 1

2 3

4

5

6 7

8

9

10

11

12 13

14

15

16

17

18

19

20

21

22

27

28

29 30

31 32

33

34

35

36

37

38 39

40

41

42

```
1
     MPI_ABORT(comm, errorcode)
2
       IN
                 comm
                                              communicator of tasks to abort
3
       IN
                 errorcode
                                              error code to return to invoking environment
4
5
6
     int MPI_Abort(MPI_Comm comm, int errorcode)
7
     MPI_Abort(comm, errorcode, ierror) BIND(C)
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          INTEGER, INTENT(IN) ::
                                      errorcode
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
11
12
     MPI_ABORT(COMM, ERRORCODE, IERROR)
          INTEGER COMM, ERRORCODE, IERROR
13
14
          This routine makes a "best attempt" to abort all tasks in the group of comm. This
15
     function does not require that the invoking environment take any action with the error
16
     code. However, a Unix or POSIX environment should handle this as a return errorcode
17
     from the main program.
18
          It may not be possible for an MPI implementation to abort only the processes repre-
19
     sented by comm if this is a subset of the processes. In this case, the MPI implementation
20
     should attempt to abort all the connected processes but should not abort any unconnected
21
     processes. If no processes were spawned, accepted, or connected then this has the effect of
22
     aborting all the processes associated with MPI_COMM_WORLD.
23
^{24}
           Rationale. The communicator argument is provided to allow for future extensions of
25
           MPI to environments with, for example, dynamic process management. In particular,
26
           it allows but does not require an MPI implementation to abort a subset of
27
           MPI_COMM_WORLD. (End of rationale.)
28
29
           Advice to users. Whether the errorcode is returned from the executable or from the
30
           MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI
31
           library but not mandatory. (End of advice to users.)
32
33
34
                                      Where possible, a high-quality implementation will try
           Advice to implementors.
35
           to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or
           singleton init). (End of advice to implementors.)
36
37
38
            Allowing User Functions at Process Termination
     8.7.1
39
     There are times in which it would be convenient to have actions happen when an MPI process
40
     finishes. For example, a routine may do initializations that are useful until the MPI job (or
41
     that part of the job that being terminated in the case of dynamically created processes) is
42
     finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF
43
     with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent
44
     of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function
45
     to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that
46
     they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then
47
     no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts
48
```

of MPI are affected. Thus, for example, calling MPI\_FINALIZED will return false in any of these callback functions. Once done with MPI\_COMM\_SELF, the order and rest of the actions taken by MPI\_FINALIZE is not specified.

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

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 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 function is needed:

```
MPI_FINALIZED(flag)
OUT flag true if MPI was finalized (logical)
int MPI_Finalized(int *flag)
MPI_Finalized(flag, ierror) BIND(C)
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_FINALIZED(FLAG, IERROR)
LOGICAL FLAG
INTEGER IERROR
```

This routine returns true if MPI\_FINALIZE has completed. It is valid to call MPI\_FINALIZED before MPI\_INIT and after MPI\_FINALIZE.

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 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 invoked during MPI\_FINALIZE. (End of advice to users.)

## 8.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

mpirun <mpirun arguments> <program> <program arguments>

Separating the command to start the program from the program itself provides flexibility, 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.

<sup>4</sup> Having a standard startup mechanism also extends the portability of MPI programs one <sup>5</sup> step further, to the command lines and scripts that manage them. For example, a validation <sup>6</sup> suite script that runs hundreds of programs can be a portable script if it is written using such <sup>7</sup> a standard starup mechanism. In order that the "standard" command not be confused with <sup>8</sup> existing practice, which is not standard and not portable among implementations, instead <sup>9</sup> of mpirun MPI specifies mpiexec.

<sup>10</sup> While a standardized startup mechanism improves the usability of MPI, the range of <sup>11</sup> environments is so diverse (e.g., there may not even be a command line interface) that MPI <sup>12</sup> cannot mandate such a mechanism. Instead, MPI specifies an mpiexec startup command <sup>13</sup> and recommends but does not require it, as advice to implementors. However, if an im-<sup>14</sup> plementation does provide a command called mpiexec, it must be of the form described <sup>15</sup> below.

<sup>16</sup> It is suggested that

17

18

23

24

25

26

28 29

30

31

32

33

34

35

36

37

38 39

40

41

42

43 44

45 46 47

48

mpiexec -n <numprocs> <program>

mpiexec -n

Form A:

-soft

-arch

-path

-file

. . .

-host <

-wdir <

Analogous to MPI\_COMM\_SPAWN, we have

<

<

<

<

<command line>

<maxprocs>

>

>

>

>

>

>

for the case where a single command line for the application program and its arguments

will suffice. See Section 10.3.4 for the meanings of these arguments. For the case

mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }

As with MPI\_COMM\_SPAWN, all the arguments are optional. (Even the -n x argu-

ment is optional; the default is implementation dependent. It might be 1, it might be

corresponding to MPI\_COMM\_SPAWN\_MULTIPLE there are two possible formats:

<sup>19</sup> be at least one way to start <program> with an initial MPI\_COMM\_WORLD whose group <sup>20</sup> contains <numprocs> processes. Other arguments to mpiexec may be implementation-<sup>21</sup> dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that mpiexec be able to be viewed as a command-line version of MPI\_COMM\_SPAWN (See Section 10.3.4).

27

362

1

 $\mathbf{2}$ 

taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the info argument to MPI\_COMM\_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

```
mpiexec -configfile <filename>
```

where the lines of <filename> are of the form separated by the colons in Form A. Lines beginning with '#' are comments, and lines may be continued by terminating the partial line with '\'.

**Example 8.11** Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

**Example 8.12** Start 10 processes on the machine called ferrari:

```
mpiexec -n 10 -host ferrari myprog
```

**Example 8.13** Start three copies of the same program with different command-line arguments:

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

**Example 8.14** Start the ocean program on five Suns and the atmos program on 10 RS/6000's:

mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

**Example 8.15** Start the ocean program on five Suns and the atmos program on 10 RS/6000's (Form B):

mpiexec -configfile myfile

where myfile contains

-n 5 -arch sun ocean -n 10 -arch rs6000 atmos

(End of advice to implementors.)

# 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, and INTEGER in Fortran with the mpi module or the deprecated 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 info is used as an argument to a nonblocking routine, it 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. Valid values for a boolean must 

<sup>16</sup> ticketWG.

 $^{24}$ 

 $^{31}$ 

1 include the strings "true" and "false" (all lowercase). For integers, valid 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 integer is a valid value 4 for a given key.) On positive numbers, + signs are optional. No space may appear between  $\mathbf{5}$ a + or - sign and the leading digit of a number. For comma separated lists, the string 6 must contain valid elements separated by commas. Leading and trailing spaces are stripped  $\overline{7}$ automatically from the types of info values described above and for each element of a comma 8 separated list. These rules apply to all info values of these types. Implementations are free 9 to specify a different interpretation for values of other info keys. 10 11MPI\_INFO\_CREATE(info) 1213OUT info info object created (handle) 1415int MPI\_Info\_create(MPI\_Info \*info) 16MPI\_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 22MPI\_INFO\_CREATE creates a new info object. The newly created object contains no 23key/value pairs.  $^{24}$ 2526MPI\_INFO\_SET(info, key, value) 27INOUT info object (handle) info 2829IN key (string) key 30 IN value value (string)  $^{31}$ 32 int MPI\_Info\_set(MPI\_Info info, const char \*key, const char \*value) 33 34MPI\_Info\_set(info, key, value, ierror) BIND(C) 35 TYPE(MPI\_Info), INTENT(IN) :: info 36 CHARACTER(LEN=\*), INTENT(IN) :: key, value 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI\_INFO\_SET(INFO, KEY, VALUE, IERROR) 39 INTEGER INFO, IERROR 40CHARACTER\*(\*) KEY, VALUE 41 42MPI\_INFO\_SET adds the (key,value) pair to info, and overrides the value if a value for 43 the same key was previously set. key and value are null-terminated strings in C. In Fortran, 44leading and trailing spaces in key and value are stripped. If either key or value are larger 45than the allowed maximums, the errors MPI\_ERR\_INFO\_KEY or MPI\_ERR\_INFO\_VALUE are

raised, respectively.

MPI_INFO	_DELETE(info, key)		1
INOUT	info	info object (handle)	2
IN	key	key (string)	3
		noj (oump)	4 5
int MPI_I	nfo_delete(MPI_Info info,	, const char *key)	6
		·	7
	_delete(info, key, ierror) MPI_Info), INTENT(IN) ::		8
	CTER(LEN=*), INTENT(IN) ::		9
	ER, OPTIONAL, INTENT(OUT)		10
			11 12
	DELETE(INFO, KEY, IERROR) ER INFO, IERROR		12
	CTER*(*) KEY		14
			15
		value) pair from info. If key is not defined in info,	16
the can rai	ises an error of class MPI_ERR.	_INFO_NOKET.	17
			18
MPI_INFO	_GET(info, key, valuelen, value	, flag)	19 20
IN	info	info object (handle)	20 21
IN	key	key (string)	22
IN	valuelen	length of value arg (integer)	23
OUT	value		24
		value (string)	25
OUT	flag	true if key defined, false if not (boolean)	26
· ·			27 28
int MPI_I	-	onst char *key, int valuelen, char *value,	20
	int *flag)		30
		value, flag, ierror) BIND(C)	31
	<pre>MPI_Info), INTENT(IN) ::</pre>		32
	CTER(LEN=*), INTENT(IN) : ER, INTENT(IN) :: value]	•	33
	CTER(LEN=valuelen), INTEN		34
	CAL, INTENT(OUT) :: flag		35 36
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
	GET(INFO, KEY, VALUELEN,	VALUE FLAC TERROR)	37 38
	ER INFO, VALUELEN, IERROF		39
	CTER*(*) KEY, VALUE	·	40
	CAL FLAG		41
This function retrieves the value associated with key in a previous call to 43			
MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,			43

MPI\_INFO\_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters available in value. If it is less than the actual size of the value, the value is truncated. In C, valuelen should be one less than the amount of allocated space to allow for the null terminator.

```
1
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
\mathbf{2}
3
     MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
4
5
                                              info object (handle)
       IN
                 info
6
       IN
                 key
                                              key (string)
7
       OUT
                 valuelen
                                              length of value arg (integer)
8
9
       OUT
                 flag
                                              true if key defined, false if not (boolean)
10
11
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
12
                     int *flag)
13
     MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) BIND(C)
14
          TYPE(MPI_Info), INTENT(IN) :: info
15
          CHARACTER(LEN=*), INTENT(IN) :: key
16
          INTEGER, INTENT(OUT) :: valuelen
17
          LOGICAL, INTENT(OUT) ::
                                       flag
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
21
          INTEGER INFO, VALUELEN, IERROR
22
          LOGICAL FLAG
23
          CHARACTER*(*) KEY
24
          Retrieves the length of the value associated with key. If key is defined, valuelen is set to
25
     the length of its associated value and flag is set to true. If key is not defined, valuelen is not
26
     touched and flag is set to false. The length returned in C does not include the end-of-string
27
     character.
28
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
29
30
^{31}
     MPI_INFO_GET_NKEYS(info, nkeys)
32
       IN
                 info
33
                                              info object (handle)
34
       OUT
                 nkeys
                                              number of defined keys (integer)
35
36
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
37
38
     MPI_Info_get_nkeys(info, nkeys, ierror) BIND(C)
          TYPE(MPI_Info), INTENT(IN) ::
39
                                              info
40
          INTEGER, INTENT(OUT) :: nkeys
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
43
          INTEGER INFO, NKEYS, IERROR
44
45
          MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
46
47
48
```

MPI_INFO_GET_NTHKEY(info, n, key) <sup>1</sup>			
IN	info	info object (handle)	2 3
IN	n	key number (integer)	4
OUT	key	key (string)	5
	,		6
int MPI_I	nfo_get_nthkey(MPI_Info i	nfo, int n, char *key)	7 8
MPI_Info_	<pre>get_nthkey(info, n, key,</pre>	ierror) BIND(C)	9
TYPE(	MPI_Info), INTENT(IN) ::	info	10
	ER, INTENT(IN) :: n		11
	CTER(LEN=*), INTENT(OUT)	•	12
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	13
	GET_NTHKEY(INFO, N, KEY,	IERROR)	14 15
	ER INFO, N, IERROR		16
CHARA	CTER*(*) KEY		17
		d key in info. Keys are numbered $0 \dots N - 1$ where	18
	÷	GET_NKEYS. All keys between 0 and $N-1$ are	19
-	l to be defined. The number o vith MPI_INFO_SET or MPI_II	f a given key does not change as long as info is not	20
modified w	THE WFI_INFO_SET OF WFI_I	VFO_DELETE.	21 22
			23
MPI_INFO	_DUP(info, newinfo)		24
IN	info	info object (handle)	25
OUT	newinfo	info object (handle)	26
			27
int MPI_I	nfo_dup(MPI_Info info, MF	PI_Info *newinfo)	28 29
MPI Info	dup(info, newinfo, ierror	) BIND(C)	30
	MPI_Info), INTENT(IN) ::		31
TYPE(	MPI_Info), INTENT(OUT) ::	newinfo	32
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	33
MPI_INFO_	DUP(INFO, NEWINFO, IERROF	2)	34
INTEG	ER INFO, NEWINFO, IERROR		35 36
MPL I	NFO DUP duplicates an exis	ting info object, creating a new object, with the	37
	value) pairs and the same orde		38
	, <b>-</b>		39
	_FREE(info)		40
			41 42
INOUT	info	info object (handle)	42
int MDT T	nfo frog(MDT Trefs with fol		44
<pre>int MPI_Info_free(MPI_Info *info)</pre>			45
			10
	free(info, ierror) BIND((		46
TYPE(	free(info, ierror) BIND(( MPI_Info), INTENT(INOUT) ER, OPTIONAL, INTENT(OUT)	:: info	

1 2	MPI_INFO_FREE(INFO, IERROR) INTEGER INFO, IERROR
3 4 5 6 7	This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is interpreted each time the info is passed to a routine. Changes to an info after return from a routine do not affect that interpretation.
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	

## Chapter 10

# **Process Creation and Management**

 $^{24}$ 

 $^{31}$ 

## 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 allows 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 latter 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. MPI 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

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

17The 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 19 between the newly created processes and the existing MPI application. It also provides a 20mechanism to establish communication between two existing MPI applications, even when one did not "start" the other. 22

23 $^{24}$ 

21

## 10.2.1 Starting Processes

25MPI applications may start new processes through an interface to an external process man-26ager.

27MPI\_COMM\_SPAWN starts MPI processes and establishes communication with them, 28returning an intercommunicator. MPI\_COMM\_SPAWN\_MULTIPLE starts several different 29binaries (or the same binary with different arguments), placing them in the same 30 MPI\_COMM\_WORLD and returning an intercommunicator.  $^{31}$ 

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

3410.2.2 The Runtime Environment 35

The MPI\_COMM\_SPAWN and MPI\_COMM\_SPAWN\_MULTIPLE routines provide an inter-36 face between MPI and the *runtime environment* of an MPI application. The difficulty is 37 that there is an enormous range of runtime environments and application requirements, and 38 39 MPI must not be tailored to any particular one. Examples of such environments are:

40 41

42

43

44

45

32

33

- 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.
- 4647
- 48

1

2

3

4 5

> 6 7

> 8

9

10

11

12

13

1415

16

- 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.

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

1 2 3 4 5 6	MPI nan of N	_COMM_WORLD tells a pre-	SIZE (See Section 10.5.1 on page 395) on ogram how "large" the initial runtime environment is, an usefully be started in all. One can subtract the size his value to find out how many processes might usefully e already running.	
7 8	10.3 F	Process Manager Inter	face	
9 10	10.3.1 F	Processes in MPI		
11 12 13 14	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 may belong to several groups.			
15	10.3.2	Starting Processes and Est	ablishing Communication	
16 17 18		ving routine starts a numbe urning an intercommunica	er of MPI processes and establishes communication with tor.	
<ol> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> </ol>	Advice to users. It is possible in MPI to start a static SPMD or MPMD appli- cation by first starting one process and having that process start its siblings with MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of perfor- mance. If possible, it is preferable to start all processes at once, as a single MPI application. ( <i>End of advice to users.</i> )			
27 28	MPI_COM	/IM_SPAWN(command, arg array_of_errcodes)	v, maxprocs, info, root, comm, intercomm,	
29 30 31	IN	command	name of program to be spawned (string, significant only at root)	
32 33	IN	argv	arguments to <b>command</b> (array of strings, significant only at root)	
34 35	IN	maxprocs	maximum number of processes to start (integer, sig- nificant only at root)	
36 37 38 39	IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, signifi- cant only at root)	
40 41	IN	root	rank of process in which previous arguments are examined (integer)	
42 43	IN	comm	intracommunicator containing group of spawning processes (handle)	
44 45 46	OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)	
46 47	OUT	array_of_errcodes	one code per process (array of integer)	

 $^{48}$ 

```
1
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
                                                                                    \mathbf{2}
              MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
                                                                                    3
              int array_of_errcodes[])
                                                                                    4
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
                                                                                    5
              array_of_errcodes, ierror) BIND(C)
                                                                                    6
    CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
                                                                                    7
    INTEGER, INTENT(IN) :: maxprocs, root
                                                                                    8
    TYPE(MPI_Info), INTENT(IN) ::
                                     info
                                                                                    9
    TYPE(MPI_Comm), INTENT(IN) ::
                                     comm
                                                                                   10
    TYPE(MPI_Comm), INTENT(OUT) ::
                                      intercomm
                                                                                   11
    INTEGER :: array_of_errcodes(*)
                                                                                   12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   13
                                                                                   14
MPI_COMM_SPAWN (COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
                                                                                   15
              ARRAY_OF_ERRCODES, IERROR)
                                                                                   16
    CHARACTER*(*) COMMAND, ARGV(*)
                                                                                   17
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
                                                                                   18
    IERROR
```

MPI\_COMM\_SPAWN tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI\_COMM\_WORLD, which is separate from that of the parents. MPI\_COMM\_SPAWN is collective over comm, and also may not return until MPI\_INIT has been called in the children. Similarly, MPI\_INIT in the children may not return until all parents have called MPI\_COMM\_SPAWN. In this sense, MPI\_COMM\_SPAWN in the parents and MPI\_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI\_COMM\_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the ordering of the group of the comm in the parents and of MPI\_COMM\_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI\_COMM\_GET\_PARENT.

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 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

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

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 9 10 that calls MPI\_INIT, the results are undefined. Implementations may allow this case to work but are not required to. 11
  - 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.)
- 18 19

The argv argument argv is an array of strings containing arguments that are passed to 2021the 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 22 NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are 23 $^{24}$ 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 and Fortran to indicate an empty argument 2526list. In C this constant is the same as NULL.

```
27
     Example 10.1 Examples of argv in C and Fortran
28
     To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
29
30
             char command[] = "ocean";
^{31}
             char *argv[] = {"-gridfile", "ocean1.grd", NULL};
32
             MPI_Comm_spawn(command, argv, ...);
33
```

```
or, if not everything is known at compile time:
```

```
35
             char *command;
36
             char **argv;
37
             command = "ocean";
38
             argv=(char **)malloc(3 * sizeof(char *));
39
             argv[0] = "-gridfile";
40
             argv[1] = "ocean1.grd";
41
             argv[2] = NULL;
42
             MPI_Comm_spawn(command, argv, ...);
43
44
     In Fortran:
45
46
47
```

48

34

1

2

3

4

5

6

7

8

12

13

14

15

16

CHARACTER*25 command, argv(3)	
command = ' ocean '	
<pre>argv(1) = ' -gridfile '</pre>	
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. 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. Second, argv of MPI\_COMM\_SPAWN must be null-terminated, so that its length can be determined.

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 382 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 and INTEGER in Fortran with the mpi module or the deprecated include file mpif.h. It is a container for a number

 $^{24}$ 

 $^{31}$ 

 $\frac{44}{45}$ 

48 ticketWG.

1 of user-specified (key,value) pairs. key and value are strings (null-terminated char\* in C,  $\mathbf{2}$ character\*(\*) in Fortran). Routines to create and manipulate the info argument are 3 described in Chapter 9 on page 365. 4 For the SPAWN calls, info provides additional (and possibly implementation-dependent)  $\mathbf{5}$ instructions to MPI and the runtime system on how to start processes. An application may 6 pass MPI\_INFO\_NULL in C or Fortran. Portable programs not requiring detailed control over 7process locations should use MPI\_INFO\_NULL. 8 MPI does not specify the content of the info argument, except to reserve a number of 9 special key values (see Section 10.3.4 on page 382). The info argument is quite flexible and 10 could even be used, for example, to specify the executable and its command-line arguments. 11In this case the **command** argument to MPI\_COMM\_SPAWN could be empty. The ability to 12do this follows from the fact that MPI does not specify how an executable is found, and the 13info argument can tell the runtime system where to "find" the executable "" (empty string). 14Of course a program that does this will not be portable across MPI implementations. 1516The root argument All arguments before the root argument are examined only on the 17process whose rank in comm is equal to root. The value of these arguments on other 18 processes is ignored. 1920The array\_of\_errcodes argument The array\_of\_errcodes is an array of length maxprocs in 21which MPI reports the status of each process that MPI was requested to start. If all maxprocs 22processes were spawned,  $\operatorname{array_of}$  errcodes is filled in with the value MPI\_SUCCESS. If only m 23 $(0 \le m \le maxprocs)$  processes are spawned, m of the entries will contain MPI\_SUCCESS and  $^{24}$ the rest will contain an implementation-specific error code indicating the reason MPI could 25not start the process. MPI does not specify which entries correspond to failed processes. 26An implementation may, for instance, fill in error codes in one-to-one correspondence with 27a detailed specification in the info argument. These error codes all belong to the error class 28MPI\_ERR\_SPAWN if there was no error in the argument list. In C or Fortran, an application 29may pass MPI\_ERRCODES\_IGNORE if it is not interested in the error codes. 30 Advice to implementors. MPI\_ERRCODES\_IGNORE in Fortran is a special type of  $^{31}$ constant, like MPI\_BOTTOM. See the discussion in Section 2.5.4 on page 15. (End of 32 advice to implementors.) 33 3435 MPI\_COMM\_GET\_PARENT(parent) 36 37 OUT the parent communicator (handle) parent 38 39 int MPI\_Comm\_get\_parent(MPI\_Comm \*parent) 4041MPI\_Comm\_get\_parent(parent, ierror) BIND(C) 42TYPE(MPI\_Comm), INTENT(OUT) :: parent INTEGER, OPTIONAL, INTENT(OUT) :: 43 ierror 44MPI\_COMM\_GET\_PARENT(PARENT, IERROR) 45INTEGER PARENT, IERROR 4647If a process was started with MPI\_COMM\_SPAWN or MPI\_COMM\_SPAWN\_MULTIPLE, 48MPI\_COMM\_GET\_PARENT returns the "parent" intercommunicator of the current process.

This parent intercommunicator is created implicitly inside of MPI\_INIT and is the same intercommunicator returned by SPAWN in the parents.

If the process was not spawned, MPI\_COMM\_GET\_PARENT returns MPI\_COMM\_NULL.

After the parent communicator is freed or disconnected, MPI\_COMM\_GET\_PARENT returns MPI\_COMM\_NULL.

Advice to users. MPI\_COMM\_GET\_PARENT returns a handle to a single intercommunicator. Calling MPI\_COMM\_GET\_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI\_COMM\_DISCONNECT or MPI\_COMM\_FREE will cause other references to the intercommunicator to become invalid (dangling). Note that calling MPI\_COMM\_FREE on the parent communicator is not useful. (*End of advice to users.*)

*Rationale.* The desire of the Forum was to create a constant MPI\_COMM\_PARENT similar to MPI\_COMM\_WORLD. Unfortunately such a constant cannot be used (syntactically) as an argument to MPI\_COMM\_DISCONNECT, which is explicitly allowed. (*End of rationale.*)

## 10.3.3 Starting Multiple Executables and Establishing Communication

While MPI\_COMM\_SPAWN is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same MPI\_COMM\_WORLD.

 $\mathbf{2}$ 

 $\overline{7}$ 

#### 1MPI\_COMM\_SPAWN\_MULTIPLE(count, array\_of\_commands, array\_of\_argv, $\mathbf{2}$ array\_of\_maxprocs, array\_of\_info, root, comm, intercomm, array\_of\_errcodes) 3 4 IN number of commands (positive integer, significant to count 5MPI only at root — see advice to users) 6 IN array\_of\_commands programs to be executed (array of strings, significant 7 only at root) 8 9 IN array\_of\_argv arguments for commands (array of array of strings, 10 significant only at root) 11IN array\_of\_maxprocs maximum number of processes to start for each com-12mand (array of integer, significant only at root) 13 IN array\_of\_info info objects telling the runtime system where and how 14to start processes (array of handles, significant only at 15root) 1617 IN rank of process in which previous arguments are exroot 18 amined (integer) 19 IN intracommunicator containing group of spawning procomm 20cesses (handle) 21OUT intercomm intercommunicator between original group and newly 22spawned group (handle) 23 $^{24}$ OUT array\_of\_errcodes one error code per process (array of integer) 2526int MPI\_Comm\_spawn\_multiple(int count, char \*array\_of\_commands[], 27char \*\*array\_of\_argv[], const int array\_of\_maxprocs[], const 28MPI\_Info array\_of\_info[], int root, MPI\_Comm comm, 29MPI\_Comm \*intercomm, int array\_of\_errcodes[]) 30 MPI\_Comm\_spawn\_multiple(count, array\_of\_commands, array\_of\_argv, $^{31}$ array\_of\_maxprocs, array\_of\_info, root, comm, intercomm, 32 array\_of\_errcodes, ierror) BIND(C) 33 INTEGER, INTENT(IN) :: count, array\_of\_maxprocs(\*), root 34 CHARACTER(LEN=\*), INTENT(IN) :: array\_of\_commands(\*) 35 CHARACTER(LEN=\*), INTENT(IN) :: array\_of\_argv(count, \*) 36 TYPE(MPI\_Info), INTENT(IN) :: array\_of\_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 42MPI\_COMM\_SPAWN\_MULTIPLE(COUNT, ARRAY\_OF\_COMMANDS, ARRAY\_OF\_ARGV, 43 ARRAY\_OF\_MAXPROCS, ARRAY\_OF\_INFO, ROOT, COMM, INTERCOMM, 44ARRAY\_OF\_ERRCODES, IERROR) 45INTEGER COUNT, ARRAY\_OF\_INFO(\*), ARRAY\_OF\_MAXPROCS(\*), ROOT, COMM, 46INTERCOMM, ARRAY\_OF\_ERRCODES(\*), IERROR 47 CHARACTER\*(\*) ARRAY\_OF\_COMMANDS(\*), ARRAY\_OF\_ARGV(COUNT, \*) 48

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.

*Rationale.* This may seem backwards to Fortran programmers who are familiar 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. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI\_COMM\_SPAWN\_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI\_ARGVS\_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (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  $m_1, m_1+1, \ldots, m_1+m_2-1$ . The processes in the third have ranks  $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 - 1$ , etc.

Advice to users. Calling MPI\_COMM\_SPAWN multiple times would create many sets of children with different MPI\_COMM\_WORLDs whereas

MPI\_COMM\_SPAWN\_MULTIPLE creates children with a single MPI\_COMM\_WORLD, 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.)

 $\overline{7}$ 

 $^{24}$ 

```
The array_of_errcodes argument is a 1-dimensional array of size \sum_{i=1}^{count} n_i, where n_i is
1
\mathbf{2}
     the i-th element of array_of_maxprocs. Command number i corresponds to the n_i contiguous
     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
3
4
     MPI_COMM_SPAWN.
5
6
     Example 10.2 Examples of array_of_argv in C and Fortran
7
     To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program
     "atmos" with argument "atmos.grd" in C:
8
9
              char *array_of_commands[2] = {"ocean", "atmos"};
10
              char **array_of_argv[2];
11
              char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
12
              char *argv1[] = {"atmos.grd", (char *)0};
13
              array_of_argv[0] = argv0;
14
              array_of_argv[1] = argv1;
15
              MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
16
17
     Here is how you do it in Fortran:
18
19
              CHARACTER*25 commands(2), array_of_argv(2, 3)
              commands(1) = ' ocean '
20
21
              array_of_argv(1, 1) = ' -gridfile '
22
              array_of_argv(1, 2) = ' ocean1.grd'
23
              array_of_argv(1, 3) = ', '
^{24}
25
              commands(2) = ' atmos '
26
              array_of_argv(2, 1) = ' atmos.grd '
27
              array_of_argv(2, 2) = ', '
28
29
              call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
30
31
     10.3.4 Reserved Keys
32
     The following keys are reserved. An implementation is not required to interpret these keys,
33
     but if it does interpret the key, it must provide the functionality described.
34
35
     host Value is a hostname. The format of the hostname is determined by the implementation.
36
37
     arch Value is an architecture name. Valid architecture names and what they mean are
38
           determined by the implementation.
39
     wdir Value is the name of a directory on a machine on which the spawned process(es)
40
           execute(s). This directory is made the working directory of the executing process(es).
41
           The format of the directory name is determined by the implementation.
42
43
     path Value is a directory or set of directories where the implementation should look for the
44
           executable. The format of path is determined by the implementation.
45
46
     file Value is the name of a file in which additional information is specified. The format of
47
           the filename and internal format of the file are determined by the implementation.
48
```

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:

by formal-50 emplets, we mean	9
1. a means $a$	10
2. <b>a:b</b> means $a, a + 1, a + 2,, b$	11
3. <b>a:b:c</b> means $a, a + c, a + 2c, \ldots, a + ck$ , where for $c > 0$ , k is the largest intege	12 r
for which $a + ck \le b$ and for $c < 0$ , k is the largest integer for which $a + ck \ge b$	) 13
If $b > a$ then c must be positive. If $b < a$ then c must be negative.	14 15
Examples:	16
1. <b>a:b</b> gives a range between $a$ and $b$	17
	18 19
2. 0:N gives full "soft" functionality	20
3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two num ber of processes.	- 21
4. 2:10000:2 allows an even number of processes.	22
5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.	23
5. 2.10.2,7 anows 2, 4, 0, 7, 8, 61 10 processes.	24 25
10.3.5 Spawn Example	26
10.5.5 Spawn Example	27
Manager-worker Example Using MPI_COMM_SPAWN	28
/* manager */	29
<pre>#include "mpi.h"</pre>	30
int main(int argc, char *argv[])	31
{	32
<pre>int world_size, universe_size, *universe_sizep, flag;</pre>	33
<pre>MPI_Comm everyone;</pre>	34
char worker_program[100];	35
	36
<pre>MPI_Init(&amp;argc, &amp;argv);</pre>	37
<pre>MPI_Comm_size(MPI_COMM_WORLD, &amp;world_size);</pre>	38
	39
<pre>if (world_size != 1) error("Top heavy with management");</pre>	40
	41
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,	42
<pre>&amp;universe_sizep, &amp;flag);</pre>	43
if (!flag) {	44
printf("This MPI does not support UNIVERSE_SIZE. How many\n\	45 46
processes total?");	40 47
<pre>scanf("%d", &amp;universe_size);</pre>	- 11

} else universe\_size = \*universe\_sizep;

 $1 \\ 2$ 

```
1
        if (universe_size == 1) error("No room to start workers");
2
3
        /*
4
         * Now spawn the workers. Note that there is a run-time determination
5
         * of what type of worker to spawn, and presumably this calculation must
6
         * be done at run time and cannot be calculated before starting
7
         * the program. If everything is known when the application is
8
         * first started, it is generally better to start them all at once
9
         * in a single MPI_COMM_WORLD.
10
         */
11
12
        choose_worker_program(worker_program);
13
        MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
14
                  MPI_INFO_NULL, 0, MPI_COMM_SELF, & everyone,
15
                  MPI_ERRCODES_IGNORE);
16
        /*
17
         * Parallel code here. The communicator "everyone" can be used
18
         * to communicate with the spawned processes, which have ranks 0,...
19
         * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
20
         * "everyone".
21
         */
22
23
        MPI_Finalize();
^{24}
        return 0;
25
     }
26
     /* worker */
27
28
     #include "mpi.h"
29
     int main(int argc, char *argv[])
30
^{31}
     ſ
32
        int size;
33
        MPI_Comm parent;
34
        MPI_Init(&argc, &argv);
        MPI_Comm_get_parent(&parent);
35
        if (parent == MPI_COMM_NULL) error("No parent!");
36
        MPI_Comm_remote_size(parent, &size);
37
        if (size != 1) error("Something's wrong with the parent");
38
39
        /*
40
41
         * Parallel code here.
         \ast The manager is represented as the process with rank 0 in (the remote
42
         * group of) the parent communicator. If the workers need to communicate
43
         * among themselves, they can use MPI_COMM_WORLD.
44
         */
45
46
47
        MPI_Finalize();
48
        return 0;
```

### }

## 10.4 Establishing Communication

This section provides functions that establish communication between two sets of MPI 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 does not 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.

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 does not 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:

- 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.

 $^{24}$ 

 $^{31}$ 

 $\frac{44}{45}$ 

386

1

2

3

4

5 6

7

8

9

10

11

12

13

25

30

31

32

33

36 37

- The server places the address information on a name server, where it can be retrieved with an agreed-upon name.
- The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

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 *system-supplied* 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 *application-supplied* service\_name so that the client could connect to that service\_name without knowing the port\_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.

- 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.
- 2. Applications that use the MPI\_PUBLISH\_NAME mechanism are completely portable 27 among implementations that provide this service. To be portable among all imple-28 mentations, these applications should have a fall-back mechanism that can be used 29 when names are not published.
  - 3. Applications may ignore MPI's name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.
- <sup>34</sup><sub>35</sub> 10.4.2 Server Routines

A server makes itself available with two routines. First it must call MPI\_OPEN\_PORT to establish a port at which it may be contacted. Secondly it must call MPI\_COMM\_ACCEPT to accept connections from clients.

- 38 39 40 41
- MPI\_OPEN\_PORT(info, port\_name)

42	IN	info	implementation-specific information on how to estab-
43			lish an address (handle)
44	OUT	port_name	newly established port (string)
45	001	p	nonity established port (corring)
46	int MDT O	pen_port(MPI_Info info, d	ther there have
47	IIIC MFI_O	pen_port(MF1_IIIO IIIO, C	liar *port_name)
48	MPI_Open_	<pre>port(info, port_name, ier</pre>	rror) BIND(C)

```
TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                       ierror
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERROR
```

This function establishes a network address, encoded in the port\_name string, at which the server will be able to accept connections from clients. port\_name is supplied by the system, possibly using information in the info argument.

MPI copies a system-supplied port name into port\_name. port\_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI\_MAX\_PORT\_NAME.

Advice to users. The system copies the port name into port\_name. The application must pass a buffer of sufficient size to hold this value. (End of advice to users.)

port\_name is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.

Advice to implementors. These examples are not meant to constrain implementations. A port\_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (End of advice to implementors.)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into an IP address. A port name may be reused after it is freed with MPI\_CLOSE\_PORT and released by the system.

Advice to implementors. Since the user may type in port\_name by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (End of advice to implementors.)

info may be used to tell the implementation how to establish the address. It may, and usually will, be MPI\_INFO\_NULL in order to get the implementation defaults.

MPI\_CLOSE\_PORT(port\_name)

IN	port_name	a port (string)
int	MPI_Close_port(const c	har *port_name)
MPI.	_Close_port(port_name, CHARACTER(LEN=*), INTE INTEGER, OPTIONAL, INT	NT(IN) :: port_name

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

2728

29

30

 $^{31}$ 

32 33

34

35

36 37

38

39 40

41 42

```
1
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
\mathbf{2}
          CHARACTER*(*) PORT_NAME
3
          INTEGER IERROR
4
     This function releases the network address represented by port_name.
5
6
\overline{7}
     MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
8
       IN
                 port_name
                                              port name (string, used only on root)
9
10
       IN
                 info
                                              implementation-dependent information (handle, used
11
                                              only on root)
12
       IN
                 root
                                              rank in comm of root node (integer)
13
       IN
                 comm
                                              intracommunicator over which call is collective (han-
14
                                              dle)
15
16
       OUT
                 newcomm
                                              intercommunicator with client as remote group (han-
17
                                              dle)
18
19
     int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
20
                     MPI_Comm comm, MPI_Comm *newcomm)
21
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) BIND(C)
22
          CHARACTER(LEN=*), INTENT(IN) :: port_name
23
          TYPE(MPI_Info), INTENT(IN) :: info
24
          INTEGER, INTENT(IN) :: root
25
          TYPE(MPI_Comm), INTENT(IN) ::
                                              comm
26
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
30
          CHARACTER*(*) PORT_NAME
^{31}
          INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
32
          MPI_COMM_ACCEPT establishes communication with a client. It is collective over the
33
     calling communicator. It returns an intercommunicator that allows communication with the
34
     client.
35
          The port_name must have been established through a call to MPI_OPEN_PORT.
36
          info can be used to provide directives that may influence the behavior of the ACCEPT
37
     call.
38
39
     10.4.3 Client Routines
40
41
     There is only one routine on the client side.
42
43
44
45
46
47
48
```

	M_CONNECT (port_name, nno		
IN	port_name	network address (string, used only on root)	2
IN	info	implementation-dependent information (handle, used only on root)	3 4 5
IN	root	rank in <b>comm</b> of root node (integer)	6
IN	comm	intracommunicator over which call is collective (han- dle)	7 8 9
OUT	newcomm	intercommunicator with server as remote group (han-dle)	9 10 11
			12
int MPI_C	omm_connect(const char *p	oort_name, MPI_Info info, int root,	13
	MPI_Comm comm, MPI_Co	omm *newcomm)	14
MPT Comm	connect(port name info	root comm newcomm jerror) BIND(C)	15
	<pre>MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror) BIND(C) CHARACTER(LEN=*), INTENT(IN) :: port_name</pre>		
	TYPE(MPI_Info), INTENT(IN) :: info INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: comm		
TYPE(			
TYPE(	MPI_Comm), INTENT(OUT) ::	newcomm	20
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	21 22
			22
	CTER*(*) PORT_NAME	noor, conn, Newconn, Thuton,	24
	ER INFO, ROOT, COMM, NEWC	COMM. IERROR	25
			26
	This routine establishes communication with a server specified by port_name. It is collective over the calling communicator and returns an intercommunicator in which the		

This routine establishes communication with a server specified by port\_name. It is collective over the calling communicator and returns an intercommunicator in which the 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.

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.*)

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

1

28

29

30

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

42

43 44

 $45 \\ 46$ 

47

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.

2 3 4

 $\mathbf{5}$ 

1

```
10.4.4 Name Publishing
```

The routines in this section provide a mechanism for publishing names. A (service\_name, 6 port\_name) pair is published by the server, and may be retrieved by a client using the 7 service\_name only. An MPI implementation defines the scope of the service\_name, that 8 is, the domain over which the service\_name can be retrieved. If the domain is the empty 9 set, that is, if no client can retrieve the information, then we say that name publishing 10 is not supported. Implementations should document how the scope is determined. High-11 quality implementations will give some control to users through the info arguments to name 12publishing functions. Examples are given in the descriptions of individual functions. 13

14

```
15
16
```

31 32

33

MPI\_PUBLISH\_NAME(service\_name, info, port\_name)

17	IN	service_name	a service name to associate with the port (string)
18	IN	info	implementation-specific information (handle)
19 20	IN	port_name	a port name (string)
21 22 23	int MP]	[_Publish_name(const char *s char *port_name)	ervice_name, MPI_Info info, const
24 25 26 27 28	TYI CH/	olish_name(service_name, inf PE(MPI_Info), INTENT(IN) :: ARACTER(LEN=*), INTENT(IN) : FEGER, OPTIONAL, INTENT(OUT)	: service_name, port_name
29 30		BLISH_NAME(SERVICE_NAME, INF FEGER INFO, IERROR	O, PORT_NAME, IERROR)

CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME

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.

<sup>40</sup> MPI permits publishing more than one service\_name for a single port\_name. On the <sup>41</sup> other hand, if service\_name has already been published within the scope determined by info, <sup>42</sup> the behavior of MPI\_PUBLISH\_NAME is undefined. An MPI implementation may, through <sup>43</sup> a mechanism in the info argument to MPI\_PUBLISH\_NAME, provide a way to allow multiple <sup>44</sup> servers with the same service in the same scope. In this case, an implementation-defined <sup>45</sup> policy will determine which of several port names is returned by MPI\_LOOKUP\_NAME.

<sup>46</sup> Note that while service\_name has a limited scope, determined by the implementation,
 <sup>47</sup> port\_name always has global scope within the communication universe used by the imple-

mentation (i.e., it is globally unique).

port\_name should be the name of a port established by MPI\_OPEN\_PORT and not yet released 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.*)

······································			
IN	service_name	a service name (string)	
IN	info	implementation-specific information (handle)	
IN	port_name	a port name (string)	
<pre>int MPI_Unpublish_name(const char *service_name, MPI_Info info, const</pre>			
<pre>MPI_Unpublish_name(service_name, info, port_name, ierror) BIND(C)     CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name     TYPE(MPI_Info), INTENT(IN) :: info     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) INTEGER INFO, IERROR			

CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME

MPI\_UNPUBLISH\_NAME(service\_name, info, port\_name)

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class MPI\_ERR\_SERVICE.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of MPI\_UNPUBLISH\_NAME is implementation dependent when a process tries to unpublish a name that it did not publish.

If the info argument was used with MPI\_PUBLISH\_NAME to tell the implementation 45 how to publish names, the implementation may require that info passed to 46 MPI\_UNPUBLISH\_NAME contain information to tell the implementation how to unpublish 47 a name. 48

 $\mathbf{2}$ 

1 MPI\_LOOKUP\_NAME(service\_name, info, port\_name) 2 IN service\_name a service name (string) 3 IN info implementation-specific information (handle) 4 5OUT port\_name a port name (string) 6  $\overline{7}$ int MPI\_Lookup\_name(const char \*service\_name, MPI\_Info info, 8 char \*port\_name) 9 MPI\_Lookup\_name(service\_name, info, port\_name, ierror) BIND(C) 10 CHARACTER(LEN=\*), INTENT(IN) :: service\_name 11 TYPE(MPI\_Info), INTENT(IN) :: info 12CHARACTER(LEN=MPI\_MAX\_PORT\_NAME), INTENT(OUT) :: port\_name 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_LOOKUP\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) 16CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME 17 INTEGER INFO, IERROR 18 This function retrieves a port\_name published by MPI\_PUBLISH\_NAME with 19service\_name. If service\_name has not been published, it raises an error in the error class 20MPI\_ERR\_NAME. The application must supply a port\_name buffer large enough to hold the 21largest possible port name (see discussion above under MPI\_OPEN\_PORT). 22 If an implementation allows multiple entries with the same service\_name within the 23same scope, a particular **port\_name** is chosen in a way determined by the implementation.  $^{24}$ If the info argument was used with MPI\_PUBLISH\_NAME to tell the implementation 2526how to publish names, a similar info argument may be required for MPI\_LOOKUP\_NAME. 2710.4.5 Reserved Key Values 2829The following key values are reserved. An implementation is not required to interpret these 30 key values, but if it does interpret the key value, it must provide the functionality described.  $^{31}$ 32 ip\_port Value contains IP port number at which to establish a port. (Reserved for 33 MPI\_OPEN\_PORT only). 3435 ip\_address Value contains IP address at which to establish a port. If the address is not a 36 valid IP address of the host on which the MPI\_OPEN\_PORT call is made, the results 37 are undefined. (Reserved for MPI\_OPEN\_PORT only). 38 39 10.4.6 Client/Server Examples 40Simplest Example — Completely Portable. 41 42The following example shows the simplest way to use the client/server interface. It does 43not use service names at all. 44On the server side: 454647char myport[MPI\_MAX\_PORT\_NAME]; 48 MPI\_Comm intercomm;

/* */	1			
<pre>MPI_Open_port(MPI_INFO_NULL, myport);</pre>				
<pre>printf("port name is: %s\n", myport);</pre>	3			
	4			
<pre>MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &amp;intercomm);</pre>	5			
/* do something with intercomm */	6 7			
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				
	11			
MPI_Comm intercomm;	12			
<pre>char name[MPI_MAX_PORT_NAME]; printf("enter port name: ");</pre>				
gets(name);	14			
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);	15			
···· <u>·</u> ······ <u>·</u> ······················	16			
Ocean/Atmosphere — Relies on Name Publishing	17			
	18			
In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere	19			
climate model. It assumes that the MPI implementation publishes names.	20 21			
	21			
MPI_Open_port(MPI_INFO_NULL, port_name);	23			
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);	24			
	25			
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);	26			
<pre>/* do something with intercomm */</pre>	27			
<pre>MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);</pre>	28			
	29			
On the client side:	30			
On the client side:	31			
<pre>MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);</pre>	32			
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,	33 34			
&intercomm);	35			
	36			
Simple Client-Server Example	37			
This is a simple example; the server accepts only a single connection at a time and serves	38			
that connection until the client requests to be disconnected. The server is a single process.	39			
Here is the server. It accepts a single connection and then processes data until it	40			
receives a message with tag 1. A message with tag 0 tells the server to exit.	41			
<i>и</i> ·	42			
<pre>#include "mpi.h" int main(int arms, chan tangu[])</pre>	43			
<pre>int main(int argc, char *argv[]) {</pre>	44			
MPI_Comm client;	45			
MPI_Status status;	46 47			
char port_name[MPI_MAX_PORT_NAME];	47			

```
1
         double buf[MAX_DATA];
\mathbf{2}
                 size, again;
         int
3
4
         MPI_Init(&argc, &argv);
5
         MPI_Comm_size(MPI_COMM_WORLD, &size);
6
         if (size != 1) error(FATAL, "Server too big");
7
         MPI_Open_port(MPI_INFO_NULL, port_name);
8
         printf("server available at %s\n", port_name);
9
         while (1) {
10
              MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
11
                                 &client);
12
              again = 1;
13
              while (again) {
14
                  MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
15
                             MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
                  switch (status.MPI_TAG) {
16
17
                       case 0: MPI_Comm_free(&client);
18
                                MPI_Close_port(port_name);
19
                                MPI_Finalize();
20
                                return 0;
21
                       case 1: MPI_Comm_disconnect(&client);
22
                                again = 0;
23
                                break;
^{24}
                       case 2: /* do something */
25
                       . . .
26
                       default:
27
                                /* Unexpected message type */
28
                                MPI_Abort(MPI_COMM_WORLD, 1);
29
                       }
30
                  }
^{31}
              }
32
     }
33
         Here is the client.
34
35
     #include "mpi.h"
36
     int main( int argc, char **argv )
37
     {
38
         MPI_Comm server;
39
         double buf[MAX_DATA];
40
         char port_name[MPI_MAX_PORT_NAME];
41
42
         MPI_Init( &argc, &argv );
43
         strcpy( port_name, argv[1] );/* assume server's name is cmd-line arg */
44
45
         MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
46
                             &server );
47
48
```

```
while (!done) {
    tag = 2; /* Action to perform */
    MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
    /* etc */
    }
MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
MPI_Comm_disconnect( &server );
MPI_Finalize();
return 0;
```

#### 10.5 Other Functionality

#### Universe Size 10.5.1

}

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 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.

MPI provides an attribute on MPI\_COMM\_WORLD, MPI\_UNIVERSE\_SIZE, that allows 22 the application to obtain this information in a portable manner. This attribute indicates 23the total number of processes that are expected. In Fortran, the attribute is the integer  $^{24}$ value. In C, the attribute is a pointer to the integer value. An application typically subtracts 25the size of MPI\_COMM\_WORLD from MPI\_UNIVERSE\_SIZE to find out how many processes it 26should spawn. MPI\_UNIVERSE\_SIZE is initialized in MPI\_INIT and is not changed by MPI. If 27defined, it has the same value on all processes of MPI\_COMM\_WORLD. MPI\_UNIVERSE\_SIZE 28is determined by the application startup mechanism in a way not specified by MPI. (The 29size 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.

47MPI\_UNIVERSE\_SIZE is assumed to have been specified when an application was started, 48 and is in essence a portable mechanism to allow the user to pass to the application (through

1

 $\mathbf{2}$ 

3

4

5

6

7 8

9

10 11 12

13 14

15

16

17

18

19

20

21

30

3132

33 34

35

36

37 38

39

40 41

42

43 44

the MPI process startup mechanism, such as mpiexec) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running, MPI\_UNIVERSE\_SIZE is not updated, and the application must find out about the change through direct communication with the runtime system.

6 7

8

30 31

32

33 34

35 36

37 38

39

## 10.5.2 Singleton MPI\_INIT

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI\_INIT. Such a process can then connect to other MPI processes using the MPI\_COMM\_ACCEPT and MPI\_COMM\_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

- Advice to implementors. To start MPI processes belonging to the same
   MPI\_COMM\_WORLD requires some special coordination. The processes must be started
   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 does not 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 processes there are by looking at the size of MPI\_COMM\_WORLD. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI\_APPNUM provides such a general mechanism. (*End of rationale.*)

### 10.5.4 Releasing Connections

Before a client and server connect, they are independent MPI applications. An error in one does not affect the other. After establishing a connection with MPI\_COMM\_CONNECT and MPI\_COMM\_ACCEPT, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- Two processes are *connected* if there is a communication path (direct or indirect) between them. More precisely:
  - 1. Two processes are connected if
    - (a) they both belong to the same communicator (inter- or intra-, including MPI\_COMM\_WORLD) or
    - (b) they have previously belonged to a communicator that was freed with MPI\_COMM\_FREE instead of MPI\_COMM\_DISCONNECT or
    - (c) they both belong to the group of the same window or filehandle.
  - 2. If A is connected to B and B to C, then A is connected to C.
- Two processes are *disconnected* (also *independent*) if they are not connected.
- By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
- Processes which are connected, but do not share the same MPI\_COMM\_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI\_COMM\_DISCONNECT.

 $\overline{7}$ 

 $^{24}$ 

 $44 \\ 45$ 

1	The following additional rules apply to MPI routines in other chapters:			
2 3	• MPI_FINALIZE is collective over a set of connected processes.			
4 5 6 7 8	• MPI_ABORT does not abort independent processes. It may abort all processes in the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.			
9 10 11 12	• If a process terminates without calling MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.			
13	MPI_COMM_DISCONNECT(comm)			
14 15 16	INOUT comm communicator (handle)			
17	<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>			
18 19 20 21	<pre>MPI_Comm_disconnect(comm, ierror) BIND(C)     TYPE(MPI_Comm), INTENT(INOUT) :: comm     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
22 23	MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR			
24 25 26	This function waits for all pending communication on <b>comm</b> to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.			
27 28 29 30	It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the			
31 32 33 34	same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.			
35 36 37 38 39 40	Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> )			
41 42 43	<i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. ( <i>End of rationale.</i> )			
44 45				
46 47				
48				

## 10.5.5 Another Way to Establish MPI Communication

MPI\_COMM\_JOIN(fd, intercomm)

IN	fd	socket file descriptor			
OUT	intercomm	new intercommunicator (handle)			
<pre>int MPI_Comm_join(int fd, MPI_Comm *intercomm) MPI_Comm_join(fd, intercomm, ierror) BIND(C)</pre>					
INTEGER, INTENT(IN) :: fd TYPE(MPI_Comm), INTENT(OUT) :: intercomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
MPI_COMM_JOIN(FD, INTERCOMM, IERROR) INTEGER FD, INTERCOMM, IERROR					

MPI\_COMM\_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [45, 49]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for MPI\_COMM\_JOIN and should return MPI\_COMM\_NULL.

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI\_COMM\_JOIN should normally succeed if the local and remote 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 byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI\_COMM\_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI\_COMM\_JOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI\_COMM\_JOIN. The two processes are referred to as the local and remote processes.

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from MPI\_COMM\_JOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an intercommunicator, but is able to leave the socket in its  $^{2}$ original state, with no pending communication, it succeeds and sets intercomm to MPI\_COMM\_NULL. The socket must be quiescent before MPI\_COMM\_JOIN is called and after  $\mathbf{5}$ MPI\_COMM\_JOIN returns. More specifically, on entry to MPI\_COMM\_JOIN, a read on the socket will not read any data that was written to the socket before the remote process called  $\overline{7}$ MPI\_COMM\_JOIN. On exit from MPI\_COMM\_JOIN, a read will not read any data that was written to the socket before the remote process returned from MPI\_COMM\_JOIN. It is the responsibility of the application to ensure the first condition, and the responsibility of the MPI implementation to ensure the second. In a multithreaded application, the application must ensure that one thread does not access the socket while another is calling MPI\_COMM\_JOIN, or call MPI\_COMM\_JOIN concurrently. Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (End of advice to implementors.) MPI\_COMM\_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI\_COMM\_JOIN on two connected processes (see Section 10.5.4 on page 397 for the definition of connected) is undefined. The returned communicator may be used to establish MPI communication with addi-tional processes, through the usual MPI communicator creation mechanisms.  $^{24}$ 

## 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, 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 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.

7 8 9

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

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. The following communication calls are provided:

- Remote write: MPI\_PUT, MPI\_RPUT
- Remote read: MPI\_GET, MPI\_RGET
- Remote update: MPI\_ACCUMULATE, MPI\_RACCUMULATE
- Remote read and update: MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP
- Remote atomic swap operations: MPI\_COMPARE\_AND\_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as "accumulate" operations.

1 MPI supports two fundamentally different memory models: separate and unified. The  $\mathbf{2}$ separate model makes no assumption about memory consistency and is highly portable. 3 This model is similar to that of weakly coherent memory systems: the user must impose 4 correct ordering of memory accesses through synchronization calls. The unified model can  $\mathbf{5}$ exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are 6 commonly available in high-performance systems. The two different models are discussed  $\overline{7}$ in detail in Section 11.4. Both models support several synchronization calls to support 8 different synchronization styles.

<sup>9</sup> The design of the RMA functions allows implementors to take advantage of fast or <sup>10</sup> asynchronous communication mechanisms provided by various platforms, such as coherent <sup>11</sup> or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and <sup>12</sup> communication coprocessors. The most frequently used RMA communication mechanisms <sup>13</sup> can be layered on top of message-passing. However, certain RMA functions might need <sup>14</sup> support for asynchronous communication agents in software (handlers, threads, etc.) in a <sup>15</sup> distributed memory environment.

<sup>16</sup> We shall denote by *origin* the process that performs the call, and by *target* the process <sup>17</sup> in which the memory is accessed. Thus, in a put operation, source=origin and destina-<sup>18</sup> tion=target; in a get operation, source=target and destination=origin.

19 20

21

## 11.2 Initialization

 $^{22}_{23}$  MPI provides the following window initialization functions: MPI\_WIN\_CREATE,

MPI\_WIN\_ALLOCATE, MPI\_WIN\_ALLOCATE\_SHARED, and

<sup>24</sup> MPI\_WIN\_CREATE\_DYNAMIC, which are collective on an intracommunicator.

<sup>25</sup> MPI\_WIN\_CREATE allows each process to specify a "window" in its memory that is made <sup>26</sup> accessible to accesses by remote processes. The call returns an opaque object that represents <sup>27</sup> the group of processes that own and access the set of windows, and the attributes of each <sup>28</sup> window, as specified by the initialization call. MPI\_WIN\_ALLOCATE differs from

<sup>29</sup> MPI\_WIN\_CREATE in that the user does not pass allocated memory;

<sup>31</sup> MPI\_WIN\_ALLOCATE returns a pointer to memory allocated by the MPI implementation. <sup>32</sup> MPI\_WIN\_ALLOCATE\_SHARED differs from MPI\_WIN\_ALLOCATE in that the allocated <sup>33</sup> memory can be accessed from all processes in the window's group with direct load/store <sup>34</sup> instructions. Some restrictions may apply to the specified communicator.

<sup>34</sup> MPI\_WIN\_CREATE\_DYNAMIC creates a window that allows the user to dynamically control <sup>35</sup> which memory is exposed by the window.

- 36
- 37 38
- 39
- 40
- $41 \\ 42$
- 43

- 45
- 46
- 47
- 48

#### Creation Window

11.2.1	Window Creation		1
			2
			3
MPI_W	IN_CREATE(base, size, disp_unit,	info, comm, win)	4 5
IN	base	initial address of window (choice)	6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	8 9
IN	info	info argument (handle)	10 11
IN	comm	intra-communicator (handle)	11
OUT	win	window object returned by the call (handle)	13
			14
int MP	I_Win_create(void *base, MPI	_Aint size, int disp_unit, MPI_Info info,	15
	MPI_Comm comm, MPI_W	-	16
TY IN IN	n_create(base, size, disp_ur PE(*), DIMENSION(), ASYNCH FEGER(KIND=MPI_ADDRESS_KIND) FEGER, INTENT(IN) :: disp_u PE(MPI_Info), INTENT(IN) ::	, INTENT(IN) :: size mit	17 18 19 20 21 22
	PE(MPI_Comm), INTENT(IN) ::		22
	PE(MPI_Win), INTENT(OUT) ::		24
IN	TEGER, OPTIONAL, INTENT(OUT)	:: ierror	25
мрт шт	N CREATE (RASE SIZE DISD II)	NIT, INFO, COMM, WIN, IERROR)	26
	<pre>vpe&gt; BASE(*)</pre>	II, INFO, CORF, WIN, IERROR/	27
	FEGER(KIND=MPI_ADDRESS_KIND)	SIZE	28
	TEGER DISP_UNIT, INFO, COMM,		29
~			30

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, 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 17.1.12 on page 630. 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, to allow windows that span more than 4 GB of address space. (Even if the physical memory size is less than 4 GB, the address range may be larger than 4 GB, if addresses are not contiguous.) (End of rationale.)

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

4243 44

45

4647

48

 $^{31}$ 

32

33

34

35

36

37

38

39

40

	404		CHAPTER	11.	ONE-SIDED COMMUNICATIONS
1 2 3 4		to byte displacem	ě		in RMA calls, and have those scaled geneous environment. ( <i>End of advice</i>
5 6 7		rgument provides window. The follo	-		the runtime about the expected usage edefined:
8 9 10 11	chronizat window.	tion (i.e., MPI_WI This implies that	N_LOCK, MPI_LC this window is n	OCK	ALL) will not be used on the given ased for 3-party communication, and conous agent activity at this process.
12 13 14		dering — controls $1.7.2$ for details.	the ordering of ac	ccum	nulate operations at the target. See
15 16 17 18 19 20 21	accumula same_op_ calls to t eliminate	ate calls to the sa no_op, then the in he same target add	me target address pplementation wil lress will use the s ect access for cert	s wi l ass same sain	tion will assume that all concurrent ll use the same operation. If set to sume that all concurrent accumulate e operation or MPI_NO_OP. This can operation types where the hardware _no_op.
22 23 24 25 26	to query is recom	the specified info	arguments window	ws t	scribed in Section 11.2.7 can be used hat have been passed to a library. It to keys for each passed window. ( <i>End</i>
27 28 29 30 31 32	windows, in lo put and accur should pose no associated with	cation, size, displanulate accesses to problem. The sa	a cement units, and a particular proc me area in memor w object. However	d inf cess ry m r, co	y specify completely different target fo arguments. As long as all the get, fit their specific target window this ay appear in multiple windows, each encurrent communications to distinct,
<ol> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> </ol>	can be a ification. process t implement	n an RMA operat target of RMA op For example, wit to use RMA opera ntation does enfor t cannot affect any	ion is to permit the perations and for the h this definition, ations, knowing the ce the specified lim	he p the i a sen nat ( nits	ry that may be accessed from another programmer to specify what memory implementation to enforce that spec- rver process can safely allow a client (under the assumption that the MPI on the exposed memory) an error in what was explicitly exposed. ( <i>End of</i>
42 43 44 45 46 47 48	MPI_ALI performa	, on some systems OC_MEM (Sectionated Sectionated Section 1997), or a section of the system of the s	, the performance n 8.2 on page 337 hen window bound	e of v 7) wi larie	in any part of the process memory. windows in memory allocated by ill be better. Also, on some systems, es are aligned at "natural" boundaries ). ( <i>End of advice to users.</i> )

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

	$\Lambda L L O C \Lambda I L (size, uisp_uint, in$	o, comm, basepti, wing	
IN	size	size of window in bytes (non-negative integer)	22
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	23 24 25
IN	info	info argument (handle)	26
IN	comm	intra-communicator (handle)	27
OUT	baseptr	initial address of window (choice)	28 29
	•		29 30
OUT	win	window object returned by the call (handle)	31
int MDT U	lin ollocoto (MDT Aint dia	e, int disp_unit, MPI_Info info,	32
IIIC MFI_W		*baseptr, MPI_Win *win)	33
		<b>•</b> •	34
	• •	info, comm, baseptr, win, ierror) BIND(C)	35
-	INTRINSIC :: ISO_C_BIND] ER(KIND=MPI_ADDRESS_KIND)	-	36
	ER, INTENT(IN) :: disp_u	-	37 38
	[MPI_Info), INTENT(IN) ::		39
	MPI_Comm), INTENT(IN) ::		40
TYPE(	C_PTR), INTENT(OUT) :: h	baseptr	41
TYPE(	MPI_Win), INTENT(OUT) ::	win	42
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	43
MPI_WIN_A	LLOCATE(SIZE, DISP_UNIT,	INFO, COMM, BASEPTR, WIN, IERROR)	44
	ER DISP_UNIT, INFO, COMM,		45
INTEG	ER(KIND=MPI_ADDRESS_KIND)	SIZE, BASEPTR	$46 \\ 47$
			47 48
			10

MPI\_WIN\_ALLOCATE(size, disp\_unit, info, comm, baseptr, win)

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11

12

13

14

15

16 17

1	This is a collective call executed by all processes in the group of <b>comm</b> . On each
2	process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
3	window object that can be used by all processes in comm to perform RMA operations. The
4	returned memory consists of size bytes local to each process, starting at address baseptr
5	and is associated with the window as if the user called MPI_WIN_CREATE on existing
6	memory. The size argument may be different at each process and $size = 0$ is valid; however, a
7	library might allocate and expose more memory in order to create a fast, globally symmetric
8	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
9	Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2
10	for an explanation of the type used for <b>baseptr</b> .
11	If the Fortran compiler provides TYPE(C_PTR), then the following interface must be
ticketWG. $^{12}$	provided in the mpi module and should be provided in the deprecated mpif.h through
13	overloading, i.e., with the same routine name as the routine with
14	INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different linker name:
15	
16	INTERFACE MPI_WIN_ALLOCATE
17	SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
18	WIN, IERROR)
19	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
20	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
21	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
22	TYPE(C_PTR) :: BASEPTR
23	END SUBROUTINE
24	END INTERFACE
25	
26	The linker name base of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The
27	The linker name base of this overloaded function is $MPI_WIN_ALLOCATE_CPTR$ . The implied linker names are described in Section 17.1.5 on page 609.
27 28	implied linker names are described in Section $17.1.5$ on page $609$ .
27 28 29	implied linker names are described in Section 17.1.5 on page 609. <i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user
27 28 29 30	<ul><li>implied linker names are described in Section 17.1.5 on page 609.</li><li><i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with</li></ul>
27 28 29 30 31	<ul><li>implied linker names are described in Section 17.1.5 on page 609.</li><li><i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory</li></ul>
27 28 29 30 31 32	<ul><li>implied linker names are described in Section 17.1.5 on page 609.</li><li><i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with</li></ul>
27 28 29 30 31 32 33	<ul><li>implied linker names are described in Section 17.1.5 on page 609.</li><li><i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be</li></ul>
27 28 29 30 31 32 33 34	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for</li> </ul>
27 28 29 30 31 32 33 34 35	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for</li> </ul>
27 28 29 30 31 32 33 34	implied linker names are described in Section 17.1.5 on page 609. <i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). ( <i>End of rationale.</i> )
27 28 29 30 31 32 33 34 35 36	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> </ul>
27 28 29 30 31 32 33 34 35 36 37	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<ul> <li>implied linker names are described in Section 17.1.5 on page 609.</li> <li>Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (End of rationale.)</li> <li>The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:</li> <li>same_size — if set to true, then the implementation may assume that the argument size is</li> </ul>

#### 11.2.3 Window That Allocates Shared Memory

MPI.	_WIN_ALLOCATE_SHARED(s	size, disp_unit, info, comm, baseptr, win)	4 5
IN	size	size of local window in bytes (non-negative integer)	6
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	7 8
IN	info	info argument (handle)	9
IN	comm	intra-communicator (handle)	10 11
οι	JT baseptr	address of local allocated window segment (choice)	12
οι	JT win	window object returned by the call (handle)	13
			14
int	MPI_Win_allocate_shared()	MPI_Aint size, int disp_unit, MPI_Info info,	15 16
	MPI_Comm comm,	void *baseptr, MPI_Win *win)	17
MPI_	Win_allocate_shared(size BIND(C)	, disp_unit, info, comm, baseptr, win, ierror)	18 19
	USE, INTRINSIC :: ISO_C	_BINDING, ONLY : C_PTR	20
		_KIND), INTENT(IN) :: size	21
	INTEGER, INTENT(IN) ::	disp_unit	22
	TYPE(MPI_Info), INTENT(I	N) :: info	23
	TYPE(MPI_Comm), INTENT(I		24
	TYPE(C_PTR), INTENT(OUT)	-	25
	TYPE(MPI_Win), INTENT(OU		26
	INTEGER, OPTIONAL, INTEN	T(OUT) :: ierror	27
MPI	WIN_ALLOCATE_SHARED(SIZE	, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	28
	INTEGER DISP_UNIT, INFO,		29 30
	INTEGER(KIND=MPI_ADDRESS		30

This is a collective call executed by all processes in the group of comm. On each 32 33 process i, it allocates memory of at least size bytes that is shared among all processes in 34comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used for load/store accesses on the calling process. The locally allocated memory can be the 35 36 target of load/store accesses by remote processes; the base pointers for other processes 37 can be queried using the function MPI\_WIN\_SHARED\_QUERY. The call also returns a window object that can be used by all processes in comm to perform RMA operations. 3839 The size argument may be different at each process and size = 0 is valid. It is the user's responsibility to ensure that the communicator comm represents a group of processes that 40 41 can create a shared memory segment that can be accessed by all processes in the group. 42The discussions of rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 8.2 also apply to MPI\_WIN\_ALLOCATE\_SHARED; in particular, see the rationale in Section 8.2 43for an explanation of the type used for **baseptr**. The allocated memory is contiguous across 44process ranks unless the info key alloc\_shared\_noncontig is specified. Contiguous across process 4546ranks means that the first address in the memory segment of process i is consecutive with 47the last address in the memory segment of process i-1. This may enable the user to 48 calculate remote address offsets with local information only.

1

1 ticketWG. 2 3 4	If the Fortran compiler provides TYPE(C_PTR), then the following interface must be provided in the mpi module and should be provided in the deprecated mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different linker name:
5 6	INTERFACE MPI_WIN_ALLOCATE_SHARED
7 8	SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, & BASEPTR, WIN, IERROR)
9	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
10	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
11	INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR
13	END SUBROUTINE
14	END INTERFACE
15	
16	The linker name base of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied linker names are described in Sec-
17 18	tion 17.1.5 on page 609.
19	The info argument can be used to specify hints similar to the info argument for
20	$MPI_WIN_CREATE,  MPI_WIN_ALLOC,  \mathrm{and}   MPI_ALLOC_MEM.$ The additional info key
21	alloc_shared_noncontig allows the library to optimize the layout of the shared memory seg-
22 23	ments in memory.
23	Advice to users. If the info key alloc_shared_noncontig is not set to true, the allocation
25	strategy is to allocate contiguous memory across process ranks. This may limit the
26	performance on some architectures because it does not allow the implementation to
27	modify the data layout (e.g., padding to reduce access latency). (End of advice to
28	users.)
29 30	Advice to implementors. If the user sets the info key alloc_shared_noncontig to true,
31	the implementation can allocate the memory requested by each process in a location
32	that is close to this process. This can be achieved by padding or allocating memory
33	in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. ( <i>End of advice to implementors.</i> )
34	consecutive ranks honeontiguous. ( <i>Linu of unified to implementors.</i> )
35 36	The consistency of load/store accesses from/to the shared memory as observed by the
37	user program depends on the architecture. A consistent view can be created in the unified
38	memory model (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling
39	MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows
40	in the separate memory model.
41 42	
43	
44	
45	
46	
47 48	
-10	

MPI_WIN	_SHARED_QUERY(win, rank,	size, disp_unit, baseptr)	1
IN	win	shared memory window object (handle)	2
IN	rank	rank in the group of window win (non-negative inte-	3
		ger) or MPI_PROC_NULL	5
OUT	size	size of the window segment (non-negative integer)	6
OUT	disp_unit	local unit size for displacements, in bytes (positive in-	7
		teger)	8
OUT	baseptr	address for load/store access to window segment	9 10
		(choice)	11
			12
int MPI_	Win_shared_query(MPI_Win	win, int rank, MPI_Aint *size,	13
	int *disp_unit, void	l *baseptr)	14
MPI_Win_	shared_query(win, rank, s	ize, disp_unit, baseptr, ierror) BIND(C)	15
	INTRINSIC :: ISO_C_BIND		16 17
TYPE	(MPI_Win), INTENT(IN) ::	win	18
	GER, INTENT(IN) :: rank		19
	GER(KIND=MPI_ADDRESS_KIND	-	20
	GER, INTENT(OUT) :: disp (C_PTR), INTENT(OUT) ::		21
	GER, OPTIONAL, INTENT(OUT	-	22
			23
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP_UNIT, IERROR			24 25
INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR			26
			27
		ocal address for remote memory segments created This function can return different process-local ad-	28
		n different processes. The returned memory can be	29
		the constraints defined in Section 11.7. This func-	30 31
tion can o	only be called with windows o	f type MPI_WIN_FLAVOR_SHARED. If the passed	32
		$OR\_SHARED$ , the error MPI_ERR_RMA_FLAVOR is	33
		, the pointer, disp_unit, and size returned are the	34
- ·	• /	ry segment belonging the lowest rank that specified $tached$ to the window specified $size = 0$ , then the	35
		MPI_ALLOC_MEM was called with size = 0.	36
	•	YPE(C_PTR), then the following interface must be	37
provided	in the mpi module and shoul	d be provided in the deprecated mpif.h through	$_{39}^{38}$ ticketWG.
	ng, i.e., with the same routin		40
INTEGER(	KIND=MPI_ADDRESS_KIND) BA	SEPTR, but with a different linker name:	41
TNTEREAC	E MPI_WIN_SHARED_QUERY		42
	·	_CPTR(WIN, RANK, SIZE, DISP_UNIT, &	43
	SEPTR, IERROR)		44
	INTRINSIC :: ISO_C_BINDI		45 46
INTEGER :: WIN, RANK, DISP_UNIT, IERROR			47
INTE	GER(KIND=MPI_ADDRESS_KIND	) :: SIZE	48

1		C_PTR) :: BASEPTR	
2		ROUTINE	
3 4	END INTER	FACE	
4 5	The li	nker name base of this overload	ed function is MPI_WIN_SHARED_QUERY_CPTR.
6		ed linker names are described i	
7	ine impire		
8 9	11.2.4 W	indow of Dynamically Attach	ed Memory
10	The MPI-2	2 RMA model requires the us	er to identify the local memory that may be a
11	target of $I$	RMA calls at the time the wa	indow is created. This has advantages for both
12			be updated by one-sided operations and provides
13	0	· · · · · ·	ation (special steps may be taken to make one-
14		°	nt). However, consider implementing a modifiable
15			w items are added to the list, memory must be is memory is typically allocated using malloc or
16			grammer must create a window with a predefined
17	-		routines for allocating memory from within the
18 19			s no easy way to handle the situation where the
20	predefined	amount of memory turns ou	t to be inadequate. To support this model, the
21			reates a window that makes it possible to expose
22	v	v	It must be used in combination with the local
23	routines M	PI_WIN_ATTACH and MPI_W	/IN_DETACH.
24			
25	MPI_WIN_	CREATE_DYNAMIC(info, com	m, win)
26 27	IN	info	info argument (handle)
28	IN	comm	intra-communicator (handle)
29	OUT	win	window object returned by the call (handle)
30			
31 32	int MPI_W	in_create_dynamic(MPI_Inf	o info, MPI_Comm comm, MPI_Win *win)
33		reate_dynamic(info, comm,	win, ierror) BIND(C)
34		MPI_Info), INTENT(IN) ::	info
35 36		MPI_Comm), INTENT(IN) ::	comm
37		<pre>MPI_Win), INTENT(OUT) :: ER, OPTIONAL, INTENT(OUT)</pre>	win :: ierror
38		ER, OFIIONAL, INIENI(001)	101101
39		REATE_DYNAMIC(INFO, COMM,	-
40	INTEG	ER INFO, COMM, WIN, IERRO	R
41	This i	s a collective call executed by	all processes in the group of comm. It returns
42		-	d. Existing process memory can be attached as
43			indow object that can be used by these processes to
44 45	-	-	mory. Because this window has special properties,
45 46	it will som		amic window
	····	etimes be referred to as a $dyn$	
47		nfo argument can be used to s	pecify hints similar to the info argument for
	The ir MPI_WIN_	nfo argument can be used to s	

In the case of a window created with MPI\_WIN\_CREATE\_DYNAMIC, the target\_disp for all RMA functions is the address at the target; i.e., the effective window\_base is MPI\_BOTTOM and the disp\_unit is one. For dynamic windows, the target\_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI\_GET\_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI\_Aint and result in unexpected values on some platforms. This issue may be addressed in a future version of MPI. (*End of advice to users.*)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI\_AINT (see Table 3.3 on page 27) is able to store addresses from any process. (End of advice to implementors.)

Memory in this window may not be used as the target of one-sided accesses in this window until it is attached using the function MPI\_WIN\_ATTACH. That is, in addition to using MPI\_WIN\_CREATE\_DYNAMIC to create an MPI window, the user must use MPI\_WIN\_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI\_WIN\_ATTACH(win, base, size)

		Dase, Size)	26
IN	win	window object (handle)	20
IN	base	initial address of memory to be attached	28
IN	size	size of memory to be attached in bytes	29
			30
int MDI	· Win attach (MD	I_Win win, void *base, MPI_Aint size)	31
IIIC MF1	WIII_attach(MF.	I_WIN WIN, VOId ≁Dase, MFI_AINU SIZe)	32
MPI_Wir	n_attach(win, ba	ase, size, ierror) BIND(C)	33
TYF	PE(MPI_Win), IN	<pre>FENT(IN) :: win</pre>	34
TYF	PE(*), DIMENSIO	N(), ASYNCHRONOUS :: base	35
INT	EGER(KIND=MPI_	ADDRESS_KIND), INTENT(IN) :: size	36
INT	EGER, OPTIONAL	, INTENT(OUT) :: ierror	37
_	MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)		
	EGER WIN, IERR	JR	40
<type> BASE(*)</type>			41
INTEGER (KIND=MPI_ADDRESS_KIND) SIZE			42

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.

 $\mathbf{5}$ 

 $\overline{7}$ 

 $45 \\ 46$ 

1	Rationale. Requ	iring that memory be explicitly attached before it is exposed to
2		y other processes can significantly simplify implementations and
3		nce. The ability to make memory available for RMA operations
4	· 0	a collective MPI_WIN_CREATE call is needed for some one-sided
5	programming mode	els. (End of rationale.)
6	Advice to users.	Attaching memory to a window may require the use of scarce
7		taching large regions of memory is not recommended in portable
8		ing memory to a window may fail if sufficient resources are not
9 10	available; this is sin	nilar to the behavior of MPI_ALLOC_MEM.
11	The user is also re	sponsible for ensuring that MPI_WIN_ATTACH at the target has
12		process attempts to target that memory with an MPI RMA call.
13	0	A operation to memory that has not been attached to a window
14	created with $MPI_{}$	WIN_CREATE_DYNAMIC is erroneous. (End of advice to users.)
15	Advice to impleme	<i>ntors.</i> A high-quality implementation will attempt to make as
16	÷	ilable for attaching as possible. Any limitations should be docu-
17		lementor. (End of advice to implementors.)
18	· · ·	
19 20	0	is a local operation as defined by MPI, which means that the call
20		pletes without requiring any MPI routine to be called in any other e detached with the routine MPI_WIN_DETACH. After memory has
22		be the target of an MPI RMA operation on that window (unless
23	, 0	ed with MPI_WIN_ATTACH).
24	the memory is re-attach	
25		
25 26	MPI_WIN_DETACH(win,	base)
26 27	MPI_WIN_DETACH(win, IN win	base) window object (handle)
26 27 28		
26 27 28 29	IN win	window object (handle)
26 27 28	IN win IN base	window object (handle)
26 27 28 29 30	IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, N	window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C)
26 27 28 29 30 31	IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IM	window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) NTENT(IN) :: win
26 27 28 29 30 31 32	<pre>IN win IN base int MPI_Win_detach(MI MPI_Win_detach(win, N TYPE(MPI_Win), IN TYPE(*), DIMENSION</pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) NTENT(IN) :: win DN(), ASYNCHRONOUS :: base</pre>
26 27 28 29 30 31 32 33 34 35	<pre>IN win IN base int MPI_Win_detach(MI MPI_Win_detach(win, N TYPE(MPI_Win), IN TYPE(*), DIMENSION</pre>	window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) NTENT(IN) :: win
26 27 28 29 30 31 32 33 34 35 36	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, N TYPE(MPI_Win), IN TYPE(*), DIMENSIC INTEGER, OPTIONAL</pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror</pre>
26 27 28 29 30 31 32 33 34 35 36 37	IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IM TYPE(*), DIMENSIC INTEGER, OPTIONAL MPI_WIN_DETACH(WIN, H	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) NTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR)</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, N TYPE(MPI_Win), IN TYPE(*), DIMENSIC INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, N INTEGER WIN, IERN</pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) NTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR)</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, N TYPE(MPI_Win), IN TYPE(*), DIMENSIG INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, N INTEGER WIN, IERN <type> BASE(*)</type></pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) WTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) ROR</pre>
26 27 28 30 31 32 33 34 35 36 37 38 39 40	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, A TYPE(MPI_Win), IN TYPE(*), DIMENSIG INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, H INTEGER WIN, IERH <type> BASE(*) Detaches a previous</type></pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) ROR</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, A TYPE(MPI_Win), IN TYPE(*), DIMENSIG INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, H INTEGER WIN, IERH <type> BASE(*) Detaches a previous</type></pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) WTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) ROR</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, A TYPE(MPI_Win), IN TYPE(*), DIMENSIC INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, H INTEGER WIN, IERH <type> BASE(*) Detaches a previous and win must match the</type></pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) ROR</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IM TYPE(*), DIMENSIG INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, M INTEGER WIN, IERM <type> BASE(*) Detaches a previous and win must match the Advice to users. In efficient use of spece</type></pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) BOR Hy attached memory region beginning at base. The arguments base arguments passed to a previous call to MPI_WIN_ATTACH. Detaching memory may permit the implementation to make more ial memory or provide memory that may be needed by a subsequent</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	<pre>IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IM TYPE(*), DIMENSIG INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, M INTEGER WIN, IERM <type> BASE(*) Detaches a previous and win must match the Advice to users. In efficient use of spece</type></pre>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) ROR ly attached memory region beginning at base. The arguments base arguments passed to a previous call to MPI_WIN_ATTACH. Detaching memory may permit the implementation to make more</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IN TYPE(*), DIMENSIO INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, H INTEGER WIN, IERN <type> BASE(*) Detaches a previous and win must match the Advice to users. In efficient use of spec MPI_WIN_ATTACH</type>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) BOR Hy attached memory region beginning at base. The arguments base arguments passed to a previous call to MPI_WIN_ATTACH. Detaching memory may permit the implementation to make more ial memory or provide memory that may be needed by a subsequent</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IN TYPE(*), DIMENSIO INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, H INTEGER WIN, IERH <type> BASE(*) Detaches a previous and win must match the Advice to users. In efficient use of spec MPI_WIN_ATTACH Memory should be</type>	window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) VTENT(IN) :: win DN(), ASYNCHRONOUS :: base ., INTENT(OUT) :: ierror BASE, IERROR) ROR Hy attached memory region beginning at base. The arguments base arguments passed to a previous call to MPI_WIN_ATTACH. Detaching memory may permit the implementation to make more ial memory or provide memory that may be needed by a subsequent 4. Users are encouraged to detach memory that is no longer needed. detached before it is freed by the user. ( <i>End of advice to users.</i> )
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	IN win IN base int MPI_Win_detach(MM MPI_Win_detach(win, M TYPE(MPI_Win), IN TYPE(*), DIMENSIO INTEGER, OPTIONAN MPI_WIN_DETACH(WIN, H INTEGER WIN, IERH <type> BASE(*) Detaches a previous and win must match the Advice to users. In efficient use of spec MPI_WIN_ATTACH Memory should be</type>	<pre>window object (handle) initial address of memory to be detached PI_Win win, const void *base) pase, ierror) BIND(C) NTENT(IN) :: win DN(), ASYNCHRONOUS :: base L, INTENT(OUT) :: ierror BASE, IERROR) ROR ly attached memory region beginning at base. The arguments base arguments passed to a previous call to MPI_WIN_ATTACH. Detaching memory may permit the implementation to make more ial memory or provide memory that may be needed by a subsequent H. Users are encouraged to detach memory that is no longer needed.</pre>

MPI\_WIN\_FREE(win)

```
INOUT
          win
                                    window object (handle)
int MPI_Win_free(MPI_Win *win)
MPI_Win_free(win, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(INOUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) ::
```

MPI\_WIN\_FREE(WIN, IERROR) INTEGER WIN, IERROR

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: 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. 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\_ALLOCATE. If the window was created with MPI\_WIN\_ALLOCATE\_SHARED, MPI\_WIN\_FREE will free the window memory that was allocated in MPI\_WIN\_ALLOCATE\_SHARED.

ierror

Freeing a window that was created with a call to MPI\_WIN\_CREATE\_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls 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 call free. This ensures 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 key to true when creating the window. In that case, an MPI implementation may free the local window without barrier synchronization. (End of advice to implementors.)

# 11.2.6 Window Attributes

The following attributes are cached with a window when the window is created.

MPI_WIN_BASE	window base address.	43
MPI_WIN_SIZE	window size, in bytes.	44
MPI_WIN_DISP_UNIT	displacement unit associated with the window.	45
MPI_WIN_CREATE_FLAVOR	how the window was created.	46
MPI_WIN_MODEL	memory model for window.	47

5

6 7

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

35

36

37

38 39

40

41 42

1 In C, calls to MPI\_Win\_get\_attr(win, MPI\_WIN\_BASE, &base, &flag),  $\mathbf{2}$ MPI\_Win\_get\_attr(win, MPI\_WIN\_SIZE, &size, &flag), 3 MPI\_Win\_get\_attr(win, MPI\_WIN\_DISP\_UNIT, &disp\_unit, &flag), 4 MPI\_Win\_get\_attr(win, MPI\_WIN\_CREATE\_FLAVOR, &create\_kind, &flag), and  $\mathbf{5}$ MPI\_Win\_get\_attr(win, MPI\_WIN\_MODEL, &memory\_model, &flag) will return in base a 6 pointer to the start of the window win, and will return in size, disp\_unit, create\_kind, and 7memory\_model pointers to the size, displacement unit of the window, the kind of routine 8 used to create the window, and the memory model, respectively. A detailed listing of the 9 type of the pointer in the attribute value argument to MPI\_WIN\_GET\_ATTR and 10 MPI\_WIN\_SET\_ATTR is shown in Table 11.1. 11Attribute C Type 12MPI\_WIN\_BASE void \* 13 MPI\_WIN\_SIZE MPI\_Aint \* 14int \* MPI\_WIN\_DISP\_UNIT 15MPI\_WIN\_CREATE\_FLAVOR int \* 1617 MPI\_WIN\_MODEL int \*18 19Table 11.1: C types of attribute value argument to MPI\_WIN\_GET\_ATTR and 20MPI\_WIN\_SET\_ATTR. 2122In Fortran, calls to MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_BASE, base, flag, ierror), 23MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_SIZE, size, flag, ierror),  $^{24}$ MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_DISP\_UNIT, disp\_unit, flag, ierror), 25MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_CREATE\_FLAVOR, create\_kind, flag, ierror), and 26MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_MODEL, memory\_model, flag, ierror) will return in 27base, size, disp\_unit, create\_kind, and memory\_model the (integer representation of) the 28base address, the size, the displacement unit of the window win, the kind of routine used to 29 create the window, and the memory model, respectively. 30 The values of create\_kind are  $^{31}$ 32 Window was created with MPI\_WIN\_CREATE. MPI\_WIN\_FLAVOR\_CREATE 33 MPI\_WIN\_FLAVOR\_ALLOCATE Window was created with 34 MPI\_WIN\_ALLOCATE. 35MPI\_WIN\_FLAVOR\_DYNAMIC Window was created with 36 MPI\_WIN\_CREATE\_DYNAMIC. 37 MPI\_WIN\_FLAVOR\_SHARED Window was created with 38 MPI\_WIN\_ALLOCATE\_SHARED. 39 The values of memory\_model are MPI\_WIN\_SEPARATE and MPI\_WIN\_UNIFIED. The mean-40 ing of these is described in Section 11.4. 41 In the case of windows created with MPI\_WIN\_CREATE\_DYNAMIC, the base address 42is MPI\_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are 43 returned, for the respective attributes. (The window attribute access functions are defined 44 in Section 6.7.3 on page 272.) The value returned for an attribute on a window is constant 45over the lifetime of the window. 46The other "window attribute," namely the group of processes attached to the window, 47

 $_{48}$  can be retrieved using the call below.

MPI_WIN_GET_GROUP(win, group)				
IN	win	window object (handle)		
OUT	group	group of processes which share access to the window (handle)		
int MPI_W	in_get_group(MPI_Win win,	MPI_Group *group)		
<pre>MPI_Win_get_group(win, group, ierror) BIND(C)    TYPE(MPI_Win), INTENT(IN) :: win    TYPE(MPI_Group), INTENT(OUT) :: group    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR) INTEGER WIN, GROUP, IERROR				

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.

### 11.2.7 Window Info

Hints specified via info (see Section 9 on page 365) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or use system resources more efficiently. However, hints do not change the semantics of any MPI interfaces. In other words, an implementation is free to ignore all hints. Hints are specified on a per window basis, in window creation functions and MPI\_WIN\_SET\_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI\_WIN\_SET\_INFO there will be no effect on previously set or default 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 the hint. (*End of advice to implementors.*)

MPI_WIN_SET_INFO(win, info)			37
	_SET_INFO(win, into)		38
INOUT	win	window object (handle)	39
IN	info	info object (handle)	40
	into	nno object (nandie)	41
int MPI_Win_set_info(MPI_Win win, MPI_Info info)			43
MPI_Win_set_info(win, info, ierror) BIND(C)			44
TYPE(MPI_Win), INTENT(IN) :: win			45
TYPE(MPI_Info), INTENT(IN) :: info			46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			47
	-		48

 $^{24}$ 

 $^{31}$ 

1	MPI_WIN	_SET_INFO(WIN, I	INFO, IERROR)	
2	INTEGER WIN, INFO, IERROR			
3	MDI	WIN SET INFO	sets new values for the hints of the window associated with win.	
4			e group of win. The info object may be different on each process,	
5			n implementation requires to be the same on all processes must	
6 7	e		e in each process's info object.	
8			1 0	
9			me info items that an implementation can use when it creates	
10			sily be changed once the window has been created. Thus, an	
11	-	· · ·	ignore hints issued in this call that it would have accepted in a	
12	cre	ation call. (End of	aavice to users.)	
13				
14				
15 16	MPI_WI	N_GET_INFO(win,	info_used)	
10	IN	win	window object (handle)	
18	OUT	info_used	new info object (handle)	
19				
20	int MPI	_Win_get_info(MI	PI_Win win, MPI_Info *info_used)	
21	MPI Win	get info(win,	info_used, ierror) BIND(C)	
22	TYPE(MPI_Win), INTENT(IN) :: win			
23 24	TYPE(MPI_Info), INTENT(OUT) :: info_used			
25	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
26	MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)			
27		EGER WIN, INFO_U		
28	MDI	WIN GET INFO	returns a new info object containing the hints of the window	
29	MPI_WIN_GET_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints actually used by the system related to			
30			info_used. If no such hints exist, a handle to a newly created	
31 32	info object is returned that contains no key/value pair. The user is responsible for freeing			
33	info_usec	I via MPI_INFO_FI	REE.	
34	4.7	· · · · · · · · · · · · · · · · · · ·		
35			e info object returned in info_used will contain all hints currently	
36			w. This set of hints may be greater or smaller than the set of the window was created, as the system may not recognize some	
37		-	and may recognize other hints that the user has not set. ( <i>End</i>	
38		advice to users.)		
39 40	5	,		
40 41	11.3	Communication		
42	11.0			

<sup>43</sup> MPI supports the following RMA communication calls: MPI\_PUT and MPI\_RPUT transfer <sup>44</sup> data from the caller memory (origin) to the target memory; MPI\_GET and MPI\_RGET <sup>45</sup> transfer data from the target memory to the caller memory; MPI\_ACCUMULATE and <sup>46</sup> MPI\_RACCUMULATE update locations in the target memory, e.g., by adding to these lo-<sup>47</sup> cations values sent from the caller memory; MPI\_GET\_ACCUMULATE,

<sup>48</sup> MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP perform atomic read-modify-write

and return the data before the accumulate operation; and MPI\_COMPARE\_AND\_SWAP performs a remote atomic 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, at the origin or both the origin and 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 on page 436. Transfers can also be completed with calls to flush routines; see Section 11.5.4 on page 448 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 on page 52.

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 operation completes at the origin.

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 the outcome of 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, the outcome of concurrent load/store and RMA updates to the same 20memory location is undefined. These restrictions are described in more detail in Section 11.7 on page 452.

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 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.)

MPI\_PROC\_NULL is a valid target rank in all MPI RMA communication calls. The effect is the same as for MPI\_PROC\_NULL in MPI point-to-point communication. 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.

## 11.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

1  $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

9

10

1112

13

1415

16

17

18

19

21

22

23 $^{24}$ 

25

26

2728

29

30

 $^{31}$ 32

33

34

35

36 37

38 39

40

41

1 MPI\_PUT(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count,  $\mathbf{2}$ target\_datatype, win) 3 IN origin\_addr initial address of origin buffer (choice) 4 IN origin\_count number of entries in origin buffer (non-negative inte-5ger) 6 7 IN origin\_datatype datatype of each entry in origin buffer (handle) 8 IN target\_rank rank of target (non-negative integer) 9 target\_disp displacement from start of window to target buffer IN 10 (non-negative integer) 11 IN target\_count number of entries in target buffer (non-negative inte-12ger) 13 14IN target\_datatype datatype of each entry in target buffer (handle) 15IN window object used for communication (handle) win 1617int MPI\_Put(const void \*origin\_addr, int origin\_count, MPI\_Datatype 18 origin\_datatype, int target\_rank, MPI\_Aint target\_disp, int 19target\_count, MPI\_Datatype target\_datatype, MPI\_Win win) 20MPI\_Put(origin\_addr, origin\_count, origin\_datatype, target\_rank, 2122 target\_disp, target\_count, target\_datatype, win, ierror) 23BIND(C) 24TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 25INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 26TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 27TYPE(MPI\_Win), INTENT(IN) :: win 28INTEGER, OPTIONAL, INTENT(OUT) :: 29 ierror 30 MPI\_PUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK,  $^{31}$ TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, IERROR) 32 <type> ORIGIN\_ADDR(\*) 33 INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 34 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 35 TARGET\_DATATYPE, WIN, IERROR 36 37 Transfers origin\_count successive entries of the type specified by the origin\_datatype, 38 starting at address origin\_addr on the origin node, to the target node specified by the win, 39 target\_rank pair. The data are written in the target buffer at address target\_addr =40window\_base + target\_disp  $\times$  disp\_unit, where window\_base and disp\_unit are the base address 41 and window displacement unit specified at window initialization, by the target process. 42The 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.

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 on page 11).

The performance of a put transfer can be significantly affected, on some systems, 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 occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)  $\mathbf{2}$ 

 $\mathbf{5}$ 

CHAPTER 11. ONE-SIDED COMMUNICATIONS

1	11.3.2	Set			
2 3					
4 5	MPI_GET	(origin_addr, origin_count, o target_datatype, win)	origin_datatype, target_rank, target_disp, target_count,		
6 7	OUT	origin_addr	initial address of origin buffer (choice)		
8 9	IN	origin_count	number of entries in origin buffer (non-negative integer)		
10	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
11 12	IN	target_rank	rank of target (non-negative integer)		
$\frac{13}{14}$	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)		
15 16	IN	target_count	number of entries in target buffer (non-negative integer)		
17 18	IN	target_datatype	datatype of each entry in target buffer (handle)		
19	IN	win	window object used for communication (handle)		
20					
21 22 23 24	MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,				
25	MPI_Datatype target_datatype, MPI_Win win)				
26 27 28	<pre>MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierror) BIND(C)</pre>				
28	TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr				
30	INTEGER, INTENT(IN) :: origin_count, target_rank, target_count				
31	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp				
32		GER(KIND-MPI_ADDRESS_K. (MPI_Win), INTENT(IN)			
33 34	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
35	MPT GFT(	ORTGIN ADDR ORIGIN COI	JNT, ORIGIN_DATATYPE, TARGET_RANK,		
36		-	ET_COUNT, TARGET_DATATYPE, WIN, IERROR)		
37	<typ< td=""><td>e&gt; ORIGIN_ADDR(*)</td><td>_ , _ , , , .</td></typ<>	e> ORIGIN_ADDR(*)	_ , _ , , , .		
38	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP				
39			IN_DATATYPE, TARGET_RANK, TARGET_COUNT,		
40 41	TARGET_DATATYPE, WIN, IERROR				
42	Simil	ar to MPI_PUT, except th	hat the direction of data transfer is reversed. Data		
43	are copied	l from the target memory	to the origin. The origin_datatype may not specify		
44			ffer. The target buffer must be contained within the		
45	0		emory in a dynamic window, and the copied data must		
46	fit, withou	it truncation, in the origin	buffer.		
47					

# 11.3.3 Examples for Communication Calls

These examples show the use of the MPI\_GET function. As all MPI RMA communication functions are nonblocking, they must be completed. In the following, this is accomplished with the routine MPI\_WIN\_FENCE, introduced in Section 11.5.

**Example 11.1** We show how to implement the generic indirect assignment A = B(map), where A, B, and map have the same distribution, and map is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER otype(p), oindex(m),
                               & ! used to construct origin datatypes
     ttype(p), tindex(m),
                             & ! used to construct target datatypes
     count(p), total(p),
                               &
     disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
! This part does the work that depends on the locations of B.
! Can be reused while this does not change
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                         &
                     comm, win, ierr)
! 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) = 0
END DO
DO i=1,m
  j = map(i)/m+1
  count(j) = count(j)+1
END DO
total(1) = 0
DO i=2,p
  total(i) = total(i-1) + count(i-1)
END DO
```

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

```
1
     DO i=1,p
\mathbf{2}
       count(i) = 0
3
     END DO
4
\mathbf{5}
     ! compute origin and target indices of entries.
6
     ! entry i at current process is received from location
\overline{7}
     ! k at process (j-1), where map(i) = (j-1)*m + (k-1),
8
     ! j = 1...p and k = 1...m
9
10
     DO i=1,m
11
       j = map(i)/m+1
12
       k = MOD(map(i), m) + 1
13
       count(j) = count(j)+1
14
       oindex(total(j) + count(j)) = i
15
       tindex(total(j) + count(j)) = k
16
     END DO
17
^{18}
     ! create origin and target datatypes for each get operation
19
     DO i=1,p
20
       CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
21
                                              oindex(total(i)+1:total(i)+count(i)), &
22
                                              MPI_REAL, otype(i), ierr)
23
       CALL MPI_TYPE_COMMIT(otype(i), ierr)
^{24}
       CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
25
                                              tindex(total(i)+1:total(i)+count(i)), &
26
                                              MPI_REAL, ttype(i), ierr)
27
       CALL MPI_TYPE_COMMIT(ttype(i), ierr)
28
     END DO
29
30
     ! this part does the assignment itself
^{31}
     CALL MPI_WIN_FENCE(0, win, ierr)
32
     disp_aint = 0
33
     DO i=1,p
34
       CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
35
     END DO
36
     CALL MPI_WIN_FENCE(0, win, ierr)
37
38
     CALL MPI_WIN_FREE(win, ierr)
39
     DO i=1,p
40
       CALL MPI_TYPE_FREE(otype(i), ierr)
41
       CALL MPI_TYPE_FREE(ttype(i), ierr)
42
     END DO
43
     RETURN
44
     END
45
46
     Example 11.2
47
48
```

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 disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                        &
                    comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  disp_aint = MOD(map(i),m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 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 contributions to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 11.7 for details.

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

1	Accumula	ate Function		
3				
4 5	MPI_AC	( <b>•</b>	origin_count, origin_datatype, target_rank, target_disp, et_datatype, op, win)	
6 7	IN	origin_addr	initial address of buffer (choice)	
8	IN	origin_count	number of entries in buffer (non-negative integer)	
9	IN	origin_datatype	datatype of each entry (handle)	
10 11	IN	target_rank	rank of target (non-negative integer)	
12 13	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	
14 15	IN	target_count	number of entries in target buffer (non-negative integer)	
16 17	IN	target_datatype	datatype of each entry in target buffer (handle)	
18	IN	ор	reduce operation (handle)	
19 20	IN	win	window object (handle)	
21 22 23 24 25 26 27 28 29 30	MPI_Acci TYPI INTI	MPI_Datatype or MPI_Aint target MPI_Datatype ta umulate(origin_addr, target_disp, ta BIND(C) E(*), DIMENSION(), EGER, INTENT(IN) ::	<pre>d *origin_addr, int origin_count, rigin_datatype, int target_rank, c_disp, int target_count, arget_datatype, MPI_Op op, MPI_Win win) origin_count, origin_datatype, target_rank, arget_count, target_datatype, op, win, ierror) INTENT(IN), ASYNCHRONOUS :: origin_addr origin_count, target_rank, target_count</pre>	
31 32 33 34 35 36	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			
30 37 38 39 40 41 42	<pre>MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>			
43 44 45 46 47 48	Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count, and origin_datatype) to the buffer specified by arguments target_count and target_datatype, at offset target_disp, in the target window specified by target_rank and win, using the operation 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.

```
22
SUBROUTINE SUM(A, B, map, m, comm, p)
                                                                                      23
USE MPI
                                                                                      ^{24}
INTEGER m, map(m), comm, p, win, ierr, disp_int
                                                                                      25
REAL A(m), B(m)
                                                                                      26
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
                                                                                      27
                                                                                      28
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
                                                                                      29
size = m * realextent
                                                                                      30
disp_int = realextent
                                                                                      31
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                           &
                                                                                      32
                      comm, win, ierr)
                                                                                      33
                                                                                      34
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                      35
DO i=1,m
                                                                                      36
  j = map(i)/m
                                                                                      37
  disp_aint = MOD(map(i),m)
                                                                                      38
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
                                                                            &
                                                                                      39
                       MPI_SUM, win, ierr)
                                                                                      40
END DO
                                                                                      41
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                      42
                                                                                      43
CALL MPI_WIN_FREE(win, ierr)
                                                                                      44
RETURN
                                                                                      45
END
                                                                                      46
```

This code is identical to the code in Example 11.2 on page 422, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the

1 2

3

4

5

6 7

8 9

10

11

12

13

14 15

16

17

18 19

20

21

47

<sup>1</sup> code computes  $B = A(map^{-1})$ , which is the reverse assignment to the one computed in <sup>2</sup> that previous example.) In a similar manner, we can replace in Example 11.1 on page 421, <sup>3</sup> the call to get by a call to accumulate, thus performing the computation with only one <sup>4</sup> communication between any two processes.

<sup>6</sup> Get Accumulate Function

<sup>8</sup> It is often useful to have fetch-and-accumulate semantics such that the remote data is <sup>9</sup> returned to the caller before the sent data is accumulated into the remote data. The get <sup>10</sup> and accumulate steps are executed atomically for each basic element in the datatype (see <sup>11</sup> Section 11.7 for details). The predefined operation MPI\_REPLACE provides fetch-and-set <sup>12</sup> behavior.

13

5

7

```
14
      MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,
15
                      result_count, result_datatype, target_rank, target_disp, target_count,
16
                     target_datatype, op, win)
17
       IN
                  origin_addr
                                               initial address of buffer (choice)
18
19
                  origin_count
       IN
                                                number of entries in origin buffer (non-negative inte-
20
                                                ger)
21
       IN
                  origin_datatype
                                               datatype of each entry in origin buffer (handle)
22
        OUT
                  result_addr
                                               initial address of result buffer (choice)
23
^{24}
       IN
                  result_count
                                               number of entries in result buffer (non-negative inte-
25
                                                ger)
26
       IN
                                               datatype of each entry in result buffer (handle)
                  result_datatype
27
       IN
                  target_rank
                                               rank of target (non-negative integer)
28
29
       IN
                  target_disp
                                                displacement from start of window to beginning of tar-
30
                                                get buffer (non-negative integer)
^{31}
       IN
                  target_count
                                                number of entries in target buffer (non-negative inte-
32
                                                ger)
33
       IN
                  target_datatype
                                                datatype of each entry in target buffer (handle)
34
35
       IN
                                                reduce operation (handle)
                  ор
36
       IN
                                                window object (handle)
                  win
37
38
      int MPI_Get_accumulate(const void *origin_addr, int origin_count,
39
                     MPI_Datatype origin_datatype, void *result_addr,
40
                      int result_count, MPI_Datatype result_datatype,
41
                      int target_rank, MPI_Aint target_disp, int target_count,
42
                     MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
43
44
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
45
                     result_count, result_datatype, target_rank, target_disp,
46
                     target_count, target_datatype, op, win, ierror) BIND(C)
47
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
48
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
```

```
INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
    target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
    result_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_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
             RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
    TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
```

Accumulate origin\_count elements of type origin\_datatype from the origin buffer ( origin\_addr) to the buffer at offset target\_disp, in the target window specified by target\_rank and win, using the operation op and return in the result buffer result\_addr the content of the target buffer before the accumulation.

The origin and result buffers (origin\_addr and result\_addr) must be disjoint. Each datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target\_datatype must not specify overlapping entries, and the target buffer must fit in the target window or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 11.7 for details.

Any of the predefined operations for MPI\_REDUCE, as well as MPI\_NO\_OP or MPI\_REPLACE can be specified as op. User-defined functions cannot be used. A new predefined operation, MPI\_NO\_OP, is defined. It corresponds to the associative function f(a,b) = a; i.e., the current value in the target memory is returned in the result buffer at the origin and no operation is performed on the target buffer. MPI\_NO\_OP can be used only in MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP. MPI\_NO\_OP cannot be used in MPI\_ACCUMULATE, MPI\_RACCUMULATE, or collective reduction operations, such as MPI\_REDUCE and others.

Advice to users. MPI\_GET is similar to MPI\_GET\_ACCUMULATE, with the operation MPI\_NO\_OP. Note, however, that MPI\_GET and MPI\_GET\_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

# Fetch and Op Function

The generic functionality of MPI\_GET\_ACCUMULATE might limit the performance of fetchand-increment or fetch-and-add calls that might be supported by special hardware operations. MPI\_FETCH\_AND\_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI\_GET\_ACCUMULATE. 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

```
428
                                         CHAPTER 11. ONE-SIDED COMMUNICATIONS
1
     MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win)
\mathbf{2}
3
                 origin_addr
       IN
                                             initial address of buffer (choice)
4
       OUT
                 result_addr
                                             initial address of result buffer (choice)
5
6
       IN
                 datatype
                                             datatype of the entry in origin, result, and target buf-
7
                                             fers (handle)
8
       IN
                 target_rank
                                             rank of target (non-negative integer)
9
                 target_disp
                                             displacement from start of window to beginning of tar-
       IN
10
                                             get buffer (non-negative integer)
11
12
       IN
                 ор
                                             reduce operation (handle)
13
       IN
                 win
                                             window object (handle)
14
15
     int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
16
                    MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
17
                    MPI_Op op, MPI_Win win)
18
19
     MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
20
                     target_disp, op, win, ierror) BIND(C)
21
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
22
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
23
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
          INTEGER, INTENT(IN) :: target_rank
25
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
26
          TYPE(MPI_Op), INTENT(IN) :: op
27
          TYPE(MPI_Win), INTENT(IN) :: win
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
29
     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
30
                    TARGET_DISP, OP, WIN, IERROR)
^{31}
          <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
32
          INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
33
          INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
34
35
          Accumulate one element of type datatype from the origin buffer (origin_addr) to the
36
     buffer at offset target_disp, in the target window specified by target_rank and win, using
37
     the operation op and return in the result buffer result_addr the content of the target buffer
38
     before the accumulation.
39
          The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the
40
     predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be
41
     specified as op; user-defined functions cannot be used. The datatype argument must be a
42
     predefined datatype. The operation is executed atomically.
43
44
     Compare and Swap Function
45
```

46 Another useful operation is an atomic compare and swap where the value at the origin is 47 compared to the value at the target, which is atomically replaced by a third value only if 48 the values at origin and target are equal.

MPI_	COMPARE_AND_SWAP(origin_ target_disp, win)	addr, compare_addr, result_addr, datatype, target_rank,	1 2
IN	origin_addr	initial address of buffer (choice)	3
IN	-		4
	compare_addr	initial address of compare buffer (choice)	5
00	T result_addr	initial address of result buffer (choice)	6 7
IN	datatype	datatype of the element in all buffers (handle)	8
IN	target_rank	rank of target (non-negative integer)	9
IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	10 11
IN	win	window object (handle)	12 13
int M		void *origin_addr, const void *compare_addr, , MPI_Datatype datatype, int target_rank, isp, MPI_Win win)	14 15 16 17
1 1 1 1 1 1 1 1 1 1 1 1	target_rank, target TYPE(*), DIMENSION(), INT TYPE(*), DIMENSION(), INT TYPE(*), DIMENSION(), ASY TYPE(MPI_Datatype), INTENT( INTEGER, INTENT(IN) :: tar INTEGER(KIND=MPI_ADDRESS_KI TYPE(MPI_Win), INTENT(IN) :: INTEGER, OPTIONAL, INTENT(C	<pre>[IN] :: datatype get_rank IND), INTENT(IN) :: target_disp : win DUT) :: ierror</pre>	18 19 20 21 22 23 24 25 26 27 28
< ]	TARGET_RANK, TARG		29 30 31 32 33
	_	nent of type datatype in the compare buffer	34 35
		et target_disp in the target window specified by	36

compare\_addr with the buffer at offset target\_disp in the target window specified by target\_rank and win and replaces the value at the target with the value in the origin buffer origin\_addr if the compare buffer and the target buffer are identical. The original value at the target is returned in the buffer result\_addr. The parameter datatype must belong to one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, Multi-language types, or Byte as specified in Section 5.9.2 on page 176. The origin and result buffers (origin\_addr and result\_addr) must be disjoint.

## 11.3.5 Request-based RMA Communication Operations

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 on page 52. Request-based RMA operations are only valid within a passive target epoch (see Section 11.5).

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	MPI_ERRO page 30). MPI_GET combinat point-to-p It is error an RMA o The MPI_WIN MPI_WIN ever, user clean up a complete	DR field in the associated All other fields of statu (	bletion call in which an RMA operation completes, the status object is set appropriately (see Section 3.2.5 on s and the results of status query functions (e.g., l. It is valid to mix different request types (e.g., any llective requests, I/O requests, generalized requests, or s that enable multiple completions (e.g., MPI_WAITALL). ST_FREE or MPI_CANCEL for a request associated with are not persistent. cplicit bulk synchronization using ISH_ALL, MPI_WIN_FLUSH_LOCAL, or lso indicates completion of the RMA operations. How- n the request handle to allow the MPI implementation to with these requests; in such cases the wait operation will nt, origin_datatype, target_rank, target_disp, target_count,	
17 18		target_datatype, wi		
19	IN	origin_addr	initial address of origin buffer (choice)	
20 21 22	IN	origin_count	number of entries in origin buffer (non-negative integer)	
23	IN	origin_datatype	datatype of each entry in origin buffer (handle)	
24	IN	target_rank	rank of target (non-negative integer)	
25 26 27	IN	target_disp	displacement from start of window to target buffer (non-negative integer)	
28 29	IN	target_count	number of entries in target buffer (non-negative integer)	
30	IN	target_datatype	datatype of each entry in target buffer (handle)	
31 32	IN	win	window object used for communication (handle)	
33	OUT	request	RMA request (handle)	
34 35 36 37 38 39 40 41 42 43	<pre>int MPI_Rput(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_Win win, MPI_Request *request) MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request, ierror) BIND(C)</pre>			
44 45 46 47 48	INTE TYPE INTE	GER, INTENT(IN) :: c (MPI_Datatype), INTEN	origin_count, target_rank, target_count NT(IN) :: origin_datatype, target_datatype KIND), INTENT(IN) :: target_disp	

	E(MPI_Request), INTENT(C EGER, OPTIONAL, INTENT(C	-	$\frac{1}{2}$	
	MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR)			
•	pe> ORIGIN_ADDR(*) EGER(KIND=MPI_ADDRESS_KI	ND) TARGET DISP	7	
		N_DATATYPE, TARGET_RANK, TARGET_COUNT,	8 9	
TAR	GET_DATATYPE, WIN, REQUE	EST, IERROR	10	
MPI	_RPUT is similar to MPI_PU	JT (Section 11.3.1), except that it allocates a commu-	11	
		es it with the request handle (the argument request).	12	
		peration (i.e., after the corresponding test or wait) in-	13 14	
		to update the locations in the origin buffer. It does le at the target window. If remote completion is re-	15	
		J_FLUSH_ALL, MPI_WIN_UNLOCK, or	16	
MPI_WI	N_UNLOCK_ALL can be use	d.	17	
			18 19	
MPI_RG	ET(origin_addr, origin_count, target_datatype, win,	origin_datatype, target_rank, target_disp, target_count, request)	20 21	
OUT	origin_addr	initial address of origin buffer (choice)	22	
IN	origin_count	number of entries in origin buffer (non-negative inte- ger)	23 24 25	
IN	origin_datatype	datatype of each entry in origin buffer (handle)	26	
IN	target_rank	rank of target (non-negative integer)	27	
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	28 29 30	
IN	target_count	number of entries in target buffer (non-negative integer)	30 31 32	
IN	target_datatype	datatype of each entry in target buffer (handle)	33	
IN	win	window object used for communication (handle)	34	
OUT	request	RMA request (handle)	35 36	
			37	
int MPI	_Rget(void *origin_addr,	int origin_count,	38	
		in_datatype, int target_rank,	39	
	-	isp, int target_count,	40 41	
	MPI_Datatype targe MPI_Request *reque	et_datatype, MPI_Win win, est)	41	
			43	
MP1_Kge		<pre>ount, origin_datatype, target_rank, et_count, target_datatype, win, request,</pre>	44	
	ierror) BIND(C)		$45 \\ 46$	
TYP	<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr</pre>			
INTECED INTENT(IN) origin count target rank target count				

INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count

		·	/			
$\frac{1}{2}$			(IN) :: origin_datatype, target_datatype			
2	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	MPI_RGET	(ORIGIN_ADDR, ORIGIN_CO	DUNT, ORIGIN_DATATYPE, TARGET_RANK,			
8			ET_COUNT, TARGET_DATATYPE, WIN, REQUEST,			
9		IERROR)				
10	• -	e> ORIGIN_ADDR(*)				
11		GER(KIND=MPI_ADDRESS_KI				
12			IN_DATATYPE, TARGET_RANK, TARGET_COUNT,			
13	IARG	ET_DATATYPE, WIN, REQUE	LSI, IERRUR			
14			ET (Section 11.3.2), except that it allocates a commu-			
15		× 0	es it with the request handle (the argument request)			
16			ompletion. The completion of an MPI_RGET operation			
17 18			in the origin buffer. If origin_addr points to memory			
19	attached	to a window, then the data	becomes available in the private copy of this window.			
20						
21	MPI_RAC	CUMULATE(origin_addr, or	igin_count, origin_datatype, target_rank, target_disp,			
22		target_count, target_c	datatype, op, win, request)			
23 24	IN	origin_addr	initial address of buffer (choice)			
24 25	IN	origin_count	number of entries in buffer (non-negative integer)			
26	IN	origin_datatype	data type of each entry in origin buffer (handle)			
27	IN	target_rank	rank of target (non-negative integer)			
28 29	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)			
30						
31 32	IN	target_count	number of entries in target buffer (non-negative integer)			
33	IN	target_datatype	datatype of each entry in target buffer (handle)			
34 35	IN	ор	reduce operation (handle)			
36	IN	win	window object (handle)			
37	OUT	request	RMA request (handle)			
38						
39	int MPI_	Raccumulate(const void	<pre>*origin_addr, int origin_count,</pre>			
40		MPI_Datatype orig	in_datatype, int target_rank,			
41 42		MPI_Aint target_d	isp, int target_count,			
42			et_datatype, MPI_Op op, MPI_Win win,			
44		MPI_Request *reque	est)			
45	MPI_Racc	umulate(origin_addr, or	rigin_count, origin_datatype, target_rank,			
46	_	•	et_count, target_datatype, op, win, request,			
47		ierror) BIND(C)	·			
48	TYPE	(*), DIMENSION(), INT	TENT(IN), ASYNCHRONOUS :: origin_addr			

<pre>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 TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR) <type> ORIGIN_ADDR(*) INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP</type></pre>			
	GER ORIGIN_COUNT, ORIGET_DATATYPE, OP, WIN,	GIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	14
			15 16
		ar to MPI_ACCUMULATE (Section 11.3.4), except that st object and associates it with the request handle (the	17
		d to wait or test for completion. The completion of an	18
		dicates that the origin buffer is free to be updated. It	19
does not	indicate that the operation	n has completed at the target window.	20 21
			21
MPI_RGE		addr, origin_count, origin_datatype, result_addr,	23
		_datatype, target_rank, target_disp, target_count,	24
	target_datatype, op		25
IN	origin_addr	initial address of buffer (choice)	26 27
IN	origin_count	number of entries in origin buffer (non-negative integer)	28 29
IN	origin_datatype	datatype of each entry in origin buffer (handle)	30
OUT	result_addr	initial address of result buffer (choice)	31
IN	result_count	number of entries in result buffer (non-negative integer)	32 33 34
IN	result_datatype	datatype of each entry in result buffer (handle)	35
IN	target_rank	rank of target (non-negative integer)	36
IN	target_disp	displacement from start of window to beginning of tar-	37 38
		get buffer (non-negative integer)	39
IN	target_count	number of entries in target buffer (non-negative integer)	40 41
IN	target_datatype	datatype of each entry in target buffer (handle)	42
IN	ор	reduce operation (handle)	43 44
IN	win	window object (handle)	44 45
OUT	request	RMA request (handle)	46
		• • /	47

434

1 int MPI\_Rget\_accumulate(const void \*origin\_addr, int origin\_count,  $\mathbf{2}$ MPI\_Datatype origin\_datatype, void \*result\_addr, 3 int result\_count, MPI\_Datatype result\_datatype, 4 int target\_rank, MPI\_Aint target\_disp, int target\_count, 5MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win, 6 MPI\_Request \*request) 7 MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 8 result\_addr, result\_count, result\_datatype, target\_rank, 9 target\_disp, target\_count, target\_datatype, op, win, request, 10 ierror) BIND(C) 11 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 12TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: result\_addr 13 INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 14target\_count 15TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype, 16result\_datatype 17 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 18 TYPE(MPI\_Op), INTENT(IN) :: op 19 TYPE(MPI\_Win), INTENT(IN) :: win 20TYPE(MPI\_Request), INTENT(OUT) :: request 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI\_RGET\_ACCUMULATE(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE,  $^{24}$ RESULT\_ADDR, RESULT\_COUNT, RESULT\_DATATYPE, TARGET\_RANK, 25TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, 26IERROR) 27<type> ORIGIN\_ADDR(\*), RESULT\_ADDR(\*) 28INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 29 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_COUNT, RESULT\_DATATYPE, 30 TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, IERROR 31MPI\_RGET\_ACCUMULATE is similar to MPI\_GET\_ACCUMULATE (Section 11.3.4), 32 except that it allocates a communication request object and associates it with the request 33 handle (the argument request) that can be used to wait or test for completion. The com-34 pletion of an MPI\_RGET\_ACCUMULATE operation indicates that the data is available in 35 the result buffer and the origin buffer is free to be updated. It does not indicate that the 36

37 38

39

40

# 11.4 Memory Model

operation has been completed at the target window.

41 The memory semantics of RMA are best understood by using the concept of public and 42private 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 43 exposed main memory in distributed memory machines). In addition, most machines have 44fast private buffers (e.g., transparent caches or explicit communication buffers) local to each 4546process where copies of data elements from the main memory can be stored for faster access. 47Such buffers are either coherent, i.e., all updates to main memory are reflected in all private 48 copies consistently, or non-coherent, i.e., conflicting accesses to main memory need to be

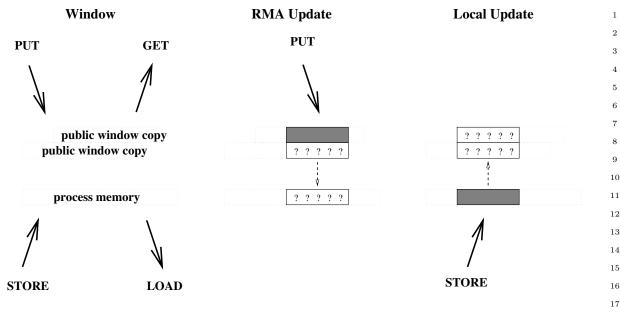


Figure 11.1: Schematic description of the public/private window operations in the MPI\_WIN\_SEPARATE memory model for two overlapping windows.

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 eventually observed by load operations without additional RMA calls. A store access to a window is eventually 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.

Advice to users. If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 11.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (End of advice to users.) 48

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 memory model for a particular RMA window can be determined by accessing the attribute MPI\_WIN\_MODEL. If the memory model is the unified model, the value of this attribute is MPI\_WIN\_UNIFIED; otherwise, the value is MPI\_WIN\_SEPARATE.

# 11.5 Synchronization Calls

RMA communications fall in two categories:

• *active target* communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

• *passive target* communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

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,

<sup>25</sup> MPI\_GET or MPI\_ACCUMULATE) on win; it completes with another synchronization call <sup>26</sup> on win. This allows users to amortize one synchronization with multiple data transfers and <sup>27</sup> provide implementors more flexibility in the implementation of RMA operations.

<sup>28</sup> Distinct access epochs for win at the same process must be disjoint. On the other hand, <sup>29</sup> epochs pertaining to different win arguments may overlap. Local operations or other MPI <sup>30</sup> calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations 32 only within an exposure epoch. Such an epoch is started and completed by RMA synchro-33 nization calls executed by the target process. Distinct exposure epochs at a process on 34the same window must be disjoint, but such an exposure epoch may overlap with exposure 35 epochs on other windows or with access epochs for the same or other win arguments. There 36 is a one-to-one matching between access epochs at origin processes and exposure epochs 37 on target processes: RMA operations issued by an origin process for a target window will 38 access that target window during the same exposure epoch if and only if they were issued 39 during the same access epoch. 40

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:

 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

1

 $\mathbf{2}$ 

3

 $\frac{4}{5}$ 

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

 $^{24}$ 

41

42

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.

3. 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 exclusive 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 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.

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 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 <sup>41</sup> synchronizations: the post occurs before the matching start, and complete occurs before <sup>42</sup> the matching wait. However, such strong synchronization is more than needed for correct <sup>43</sup> ordering of window accesses. The semantics of MPI calls allow *weak synchronization*, as <sup>44</sup> 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 <sup>46</sup> also terminate earlier, if put data is buffered by the implementation. The synchronization <sup>47</sup>

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

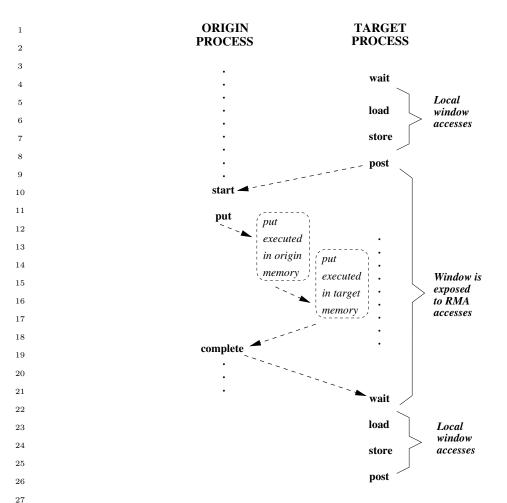


Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

29 30

28

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.

39 40

41

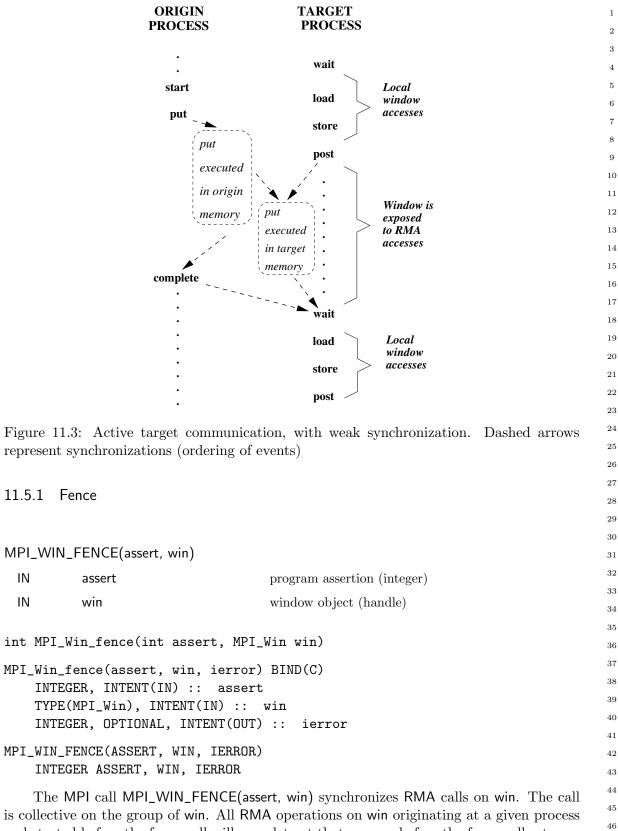
42

Rationale. RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End* of rationale.)

- 43 44
- 45
- 46
- 47
- 48

IN

IN



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

47

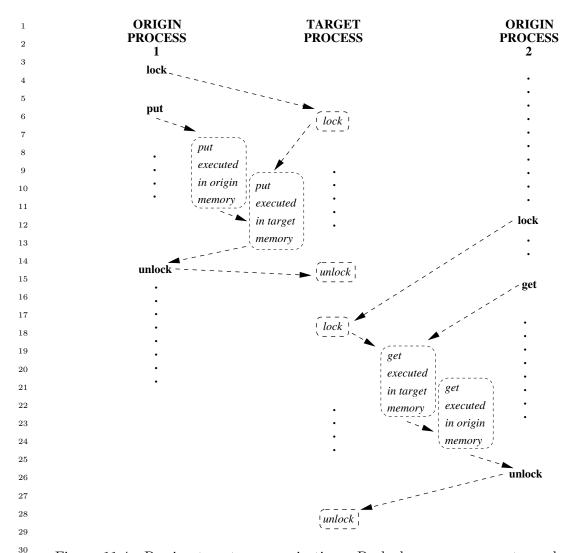


Figure 11.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

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 36 the local process issued RMA communication calls on win between these two calls. The call 37 completes an RMA exposure epoch if it was preceded by another fence call and the local 38 window was the target of RMA accesses between these two calls. The call starts an RMA 39 access epoch if it is followed by another fence call and by RMA communication calls issued 40 between these two fence calls. The call starts an exposure epoch if it is followed by another 41 fence call and the local window is the target of RMA accesses between these two fence calls. 42Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait. 43

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 equal to MPI\_MODE\_NOPRECEDE) does not necessarily act as a barrier.

33

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.

Advice to users. Calls to MPI\_WIN\_FENCE should both precede and follow calls to 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)

int MPI\_Win\_start(MPI\_Group group, int assert, MPI\_Win win)
MPI\_Win\_start(group, assert, win, ierror) BIND(C)
 TYPE(MPI\_Group), INTENT(IN) :: group
 INTEGER, INTENT(IN) :: assert
 TYPE(MPI\_Win), INTENT(IN) :: win
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
INTEGER GROUP, ASSERT, WIN, IERROR
```

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 valid.

MPI\_WIN\_COMPLETE(WIN, IERROR)

 $\mathbf{2}$ 

INTEGER WIN, IERROR 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. Example 11.4 MPI\_Win\_start(group, flag, win); MPI\_Put(..., win); MPI\_Win\_complete(win); 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 occurs; 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 has called MPI\_WIN\_POST — the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI\_WIN\_POST is issued, the sequence above must complete, without further dependencies. MPI\_WIN\_POST(group, assert, win) IN group of origin processes (handle) group IN assert program assertion (integer) IN win window object (handle) int MPI\_Win\_post(MPI\_Group group, int assert, MPI\_Win win) MPI\_Win\_post(group, assert, win, ierror) BIND(C) TYPE(MPI\_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: assert TYPE(MPI\_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_WIN\_POST(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR Starts an RMA exposure epoch for the local window associated with win. Only processes

4647in group should access the window with RMA calls on win during this epoch. Each process 48in group must issue a matching call to MPI\_WIN\_START. MPI\_WIN\_POST does not block.

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9 10

1112

1314

15

16

17

18

19

20

21

22

23

24

25

26

27

282930

 $^{31}$ 

32

33

34

35 36

37 38

39

40

41

42

43

44

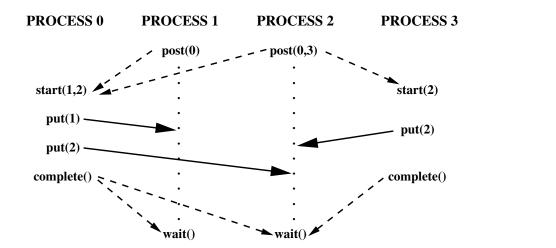


Figure 11.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

Completes an RMA exposure epoch started by a call to MPI\_WIN\_POST on win. This call matches calls to MPI\_WIN\_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI\_WIN\_WAIT will block until all matching calls to MPI\_WIN\_COMPLETE have occurred. This guarantees that all these 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.

Figure 11.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

MPI\_WIN\_TEST(win, flag)

IN	win	window object (handle)
OUT	flag	success flag (logical)

int MPI\_Win\_test(MPI\_Win win, int \*flag)

```
1
     MPI_Win_test(win, flag, ierror) BIND(C)
\mathbf{2}
          TYPE(MPI_Win), INTENT(IN) ::
                                             win
3
          LOGICAL, INTENT(OUT) :: flag
4
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
5
     MPI_WIN_TEST(WIN, FLAG, IERROR)
6
          INTEGER WIN. IERROR
7
          LOGICAL FLAG
8
9
          This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses
10
      to the local window by the group to which it was exposed by the corresponding
11
      MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE
12
     calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned
13
      immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the
14
      effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible
15
      effect.
16
          MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once
17
      the call has returned flag = true, it must not be invoked anew, until the window is posted
18
      anew.
19
          Assume that window win is associated with a "hidden" communicator wincomm, used
20
      for communication by the processes of win. The rules for matching of post and start calls
21
      and for matching complete and wait calls can be derived from the rules for matching sends
22
      and receives, by considering the following (partial) model implementation.
23
      MPI_WIN_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process
^{24}
           in group, using wincomm. There is no need to wait for the completion of these sends.
25
26
      MPI_WIN_START(group.0.win) initiates a nonblocking receive with tag tag0 from each
27
           process in group, using wincomm. An RMA access to a window in target process i is
28
           delayed until the receive from i is completed.
29
30
      MPI_WIN_COMPLETE(win) initiate a nonblocking send with tag tag1 to each process
^{31}
           in the group of the preceding start call. No need to wait for the completion of these
32
           sends.
33
      MPI_WIN_WAIT(win) initiate a nonblocking receive with tag tag1 from each process in
34
           the group of the preceding post call. Wait for the completion of all receives.
35
36
          No races can occur in a correct program: each of the sends matches a unique receive,
37
      and vice versa.
38
39
           Rationale. The design for general active target synchronization requires the user to
40
           provide complete information on the communication pattern, at each end of a com-
41
           munication link: each origin specifies a list of targets, and each target specifies a list
42
           of origins. This provides maximum flexibility (hence, efficiency) for the implementor:
43
           each synchronization can be initiated by either side, since each "knows" the identity
44
           of the other. This also provides maximum protection from possible races. On the
45
           other hand, the design requires more information than RMA needs: in general, it is
46
           sufficient for the origin to know the rank of the target, but not vice versa. Users
47
           that want more "anonymous" communication will be required to use the fence or lock
48
           mechanisms. (End of rationale.)
```

1 Advice to users. Assume a communication pattern that is represented by a directed graph  $G = \langle V, E \rangle$ , where  $V = \{0, \dots, n-1\}$  and  $ij \in E$  if origin process i accesses 2 3 the window at target process i. Then each process i issues a call to 4  $MPI_WIN_POST(ingroup_i, ...)$ , followed by a call to MPI\_WIN\_START( $outgroup_i,\ldots$ ), where  $outgroup_i = \{j : ij \in E\}$  and  $ingroup_i = \{j : ij \in E\}$ 5 $\{j : ji \in E\}$ . A call is a noop, and can be skipped, if the group argument is empty. 6  $\overline{7}$ After the communications calls, each process that issued a start will issue a complete. 8 Finally, each process that issued a post will issue a wait. 9 Note that each process may call with a group argument that has different members. 10 (End of advice to users.) 11 1211.5.3 Lock 13 1415MPI\_WIN\_LOCK(lock\_type, rank, assert, win) 1617 IN lock\_type either MPI\_LOCK\_EXCLUSIVE or 18 MPI\_LOCK\_SHARED (state) 19 IN rank rank of locked window (non-negative integer) 20IN program assertion (integer) assert 2122 IN window object (handle) win 23 $^{24}$ int MPI\_Win\_lock(int lock\_type, int rank, int assert, MPI\_Win win) 25MPI\_Win\_lock(lock\_type, rank, assert, win, ierror) BIND(C) 26INTEGER, INTENT(IN) :: lock\_type, rank, assert 27TYPE(MPI\_Win), INTENT(IN) :: win 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI\_WIN\_LOCK(LOCK\_TYPE, RANK, ASSERT, WIN, IERROR) 31INTEGER LOCK\_TYPE, RANK, ASSERT, WIN, IERROR 32 Starts an RMA access epoch. Only the window at the process with rank rank can be 33 accessed by RMA operations on win during that epoch. 34 35 36 MPI\_WIN\_LOCK\_ALL(assert, win) 37 IN assert program assertion (integer) 38 39 IN win window object (handle) 40 41 int MPI\_Win\_lock\_all(int assert, MPI\_Win win) 42MPI\_Win\_lock\_all(assert, win, ierror) BIND(C) 43 INTEGER, INTENT(IN) :: assert 44 TYPE(MPI\_Win), INTENT(IN) :: win 45INTEGER, OPTIONAL, INTENT(OUT) :: 46ierror 47MPI\_WIN\_LOCK\_ALL(ASSERT, WIN, IERROR) 48

	440		CHAPTER II. ONE-SIDED COMMUN	NICATIONS
1	INT	EGER ASSERT, WIN,	IERROR	
2 3 4 5 6 7	MPI_LOC all proces must be	K_SHARED. During t sses in win by using F unlocked with MPI_'	poch to all processes in win, with a lock type of he epoch, the calling process can access the window RMA operations. A window locked with MPI_WIN_ WIN_UNLOCK_ALL. This routine is not collective rs of the group of the window.	/ memory on _LOCK_ALL
8 9 10 11 12 13	MF ove	PI_WIN_LOCK and N erheads could be ave	re may be additional overheads associated with u PI_WIN_LOCK_ALL concurrently on the same win ided by specifying the assertion MPI_MODE_NOC .5.5). ( <i>End of advice to users.</i> )	ndow. These
14 15	MPI_WI	N_UNLOCK(rank, wii	n)	
16	IN	rank	rank of window (non-negative integer)	
17 18	IN	win	window object (handle)	
19 20	int MPI	_Win_unlock(int r	ank, MPI_Win win)	
21 22 23 24	MPI_Win_unlock(rank, win, ierror) BIND(C) INTEGER, INTENT(IN) :: rank TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
25 26 27		_UNLOCK(RANK, WIN EGER RANK, WIN, II	-	
28 29 30 31	operation		ss epoch started by a call to MPI_WIN_LOCK( period will have completed both at the origin and a	,
32 33	MPI_WI	N_UNLOCK_ALL(wir	)	
34 35	IN	win	window object (handle)	
36 37	int MPI	_Win_unlock_all(M	PI_Win win)	
38 39 40	TYP	_unlock_all(win, E(MPI_Win), INTEN EGER, OPTIONAL, I	Γ(IN) :: win	
41 42 43		_UNLOCK_ALL(WIN, EGER WIN, IERROR	IERROR)	
44 45 46 47	win). RM at the ta Lock	A operations issued rget when the call re as are used to protect	t accesses to the locked target window effected by	e origin and y RMA calls
48	issued be	etween the lock and u	nlock calls, and to protect load/store accesses to a	locked local

CHAPTER 11. ONE-SIDED COMMUNICATIONS

or shared memory window executed between the lock and unlock calls. 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. For example, a process may not call MPI\_WIN\_LOCK to lock a target window if the target process has called MPI\_WIN\_POST and has not yet called MPI\_WIN\_WAIT; it is erroneous to call MPI\_WIN\_POST while the local window is locked.

*Rationale.* An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (End of advice to users.)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI\_ALLOC\_MEM (Section 8.2 on page 337), MPI\_WIN\_ALLOCATE (Section 11.2.2 on page 405), or attached with MPI\_WIN\_ATTACH (Section 11.2.4 on page 410). Locks can be used portably only in such memory.

*Rationale.* The implementation of passive target communication when memory is not shared may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for third party communication in shared memory machines.

The downside of this decision is that passive target communication cannot be used without taking advantage of nonstandard Fortran features: namely, the availability of C-like pointers; these are not supported by some Fortran compilers. (*End of rationale.*)

Consider the sequence of calls in the example below.

#### Example 11.5

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);
MPI_Put(..., rank, ..., win);
MPI_Win_unlock(rank, win);
```

The call to MPI\_WIN\_UNLOCK will not return until the put transfer has completed at the origin and at the target. This still leaves much freedom to implementors. The call to MPI\_WIN\_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI\_WIN\_UNLOCK blocks until a lock is acquired — the 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
     update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs.
\mathbf{2}
      However, if the call to MPI_WIN_LOCK is used to lock a local window, then the call must
3
      block until the lock is acquired, since the lock may protect local load/store accesses to the
4
      window issued after the lock call returns.
\mathbf{5}
6
     11.5.4 Flush and Sync
\overline{7}
      All flush and sync functions can be called only within passive target epochs.
8
9
10
      MPI_WIN_FLUSH(rank, win)
11
        IN
                  rank
                                               rank of target window (non-negative integer)
12
13
        IN
                                               window object (handle)
                  win
14
15
     int MPI_Win_flush(int rank, MPI_Win win)
16
     MPI_Win_flush(rank, win, ierror) BIND(C)
17
          INTEGER, INTENT(IN) ::
                                       rank
18
          TYPE(MPI_Win), INTENT(IN) ::
19
                                              win
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
20
21
     MPI_WIN_FLUSH(RANK, WIN, IERROR)
22
          INTEGER RANK, WIN, IERROR
23
          MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling
^{24}
      process to the target rank on the specified window. The operations are completed both at
25
26
      the origin and at the target.
27
28
      MPI_WIN_FLUSH_ALL(win)
29
        IN
                                               window object (handle)
30
                  win
^{31}
32
      int MPI_Win_flush_all(MPI_Win win)
33
     MPI_Win_flush_all(win, ierror) BIND(C)
34
          TYPE(MPI_Win), INTENT(IN) :: win
35
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
36
37
     MPI_WIN_FLUSH_ALL(WIN, IERROR)
38
          INTEGER WIN, IERROR
39
          All RMA operations issued by the calling process to any target on the specified window
40
      prior to this call and in the specified window will have completed both at the origin and at
41
      the target when this call returns.
42
43
44
45
46
47
48
```

CHAPTER 11. ONE-SIDED COMMUNICATIONS

MPI_WIN_FLUSH_LOCAL(rank, win)			
IN r	rank	rank of target window (non-negative integer)	2
IN v	win	window object (handle)	3 4
			5
int MPI_Win	n_flush_local(int rank,	MPI_Win win)	6
MPI Win flu	ush_local(rank, win, ier	ror) BIND(C)	7
	R, INTENT(IN) :: rank		8
TYPE(MP	PI_Win), INTENT(IN) ::	win	9 10
INTEGER	R, OPTIONAL, INTENT(OUT)	:: ierror	11
MPI_WIN_FLU	JSH_LOCAL(RANK, WIN, IEF	ROR)	12
	R RANK, WIN, IERROR		13
Locally of	completes at the origin all o	utstanding RMA operations initiated by the calling	14
÷		rank on the specified window. For example, after	15 16
-	· · ·	se any buffers provided to put, get, or accumulate	16
operations.			18
			19
MPI_WIN_FL	_USH_LOCAL_ALL(win)		20
	vin	window object (handle)	21
114 V		window object (nandie)	22
int MPT Win	n_flush_local_all(MPI_Wi	n win)	23 24
			25
	ush_local_all(win, ierro PI_Win), INTENT(IN) ::		26
	R, OPTIONAL, INTENT(OUT)		27
			28
	JSH_LOCAL_ALL(WIN, IERRO	)R)	29
	R WIN, IERROR		30 31
		target prior to this call in this window will have	32
completed at	the origin when MPI_WIN_	FLUSH_LOCAL_ALL returns.	33
			34
MPI_WIN_SY	YNC(win)		35
IN v	win	window object (handle)	36
			37 38
int MPI_Win	n_sync(MPI_Win win)		39
MPT Win swn	nc(win, ierror) BIND(C)		40
•	PI_Win), INTENT(IN) ::	win	41
	R, OPTIONAL, INTENT(OUT)		42
MDT WIN GVN	NC(WIN, IERROR)		43
	WIN, IERROR		$44 \\ 45$
	-		45 46
The call MPI_WIN_SYNC synchronizes the private and public window copies of win.			47
For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the			48

effect of ending and reopening an access and exposure epoch on the window (note that it  $\mathbf{2}$ does not actually end an epoch or complete any pending MPI RMA operations).

3 4

5

1

### 11.5.5 Assertions

The assert argument in the calls MPI\_WIN\_POST, MPI\_WIN\_START, MPI\_WIN\_FENCE, 6 MPI\_WIN\_LOCK, and MPI\_WIN\_LOCK\_ALL is used to provide assertions on the context of 7 the call that may be used to optimize performance. The assert argument does not change 8 program semantics if it provides correct information on the program — it is erroneous to 9 provide incorrect information. Users may always provide assert = 0 to indicate a general 10 case where no guarantees are made. 11

1213

14

15

16

17

18 19

20

21

22

35

36

37

38

39

40

41

42

43

44

45

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's 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.)

- 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.)
- 23assert is the bit-vector OR of zero or more of the following integer constants:  $^{24}$ MPI\_MODE\_NOCHECK, MPI\_MODE\_NOSTORE, MPI\_MODE\_NOPUT,
- 25MPI\_MODE\_NOPRECEDE, and MPI\_MODE\_NOSUCCEED. The significant options are listed 26below for each call. 27
- 28Advice to users. C/C++ users can use bit vector or () to combine these constants; 29 Fortran 90 users can use the bit-vector IOR intrinsic. Fortran 77 users can use (non-30 portably) bit vector IOR on systems that support it. Alternatively, Fortran users can 31portably use integer addition to OR the constants (each constant should appear at 32 most once in the addition!). (End of advice to users.) 33

#### 34**MPI\_WIN\_START:** MPI\_MODE\_NOCHECK — the matching calls to

MPI\_WIN\_POST have already completed 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.)

# **MPI\_WIN\_POST:** MPI\_MODE\_NOCHECK — the matching calls to

- MPI\_WIN\_START have not yet occurred 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.
- 46MPI\_MODE\_NOSTORE — the local window was not updated by stores (or local get 47 or receive calls) since last synchronization. This may avoid the need for cache 48 synchronization at the post call.

- 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.
- **MPI\_WIN\_FENCE:** MPI\_MODE\_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization.
  - MPI\_MODE\_NOPUT the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
  - 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.
  - 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.
- MPI\_WIN\_LOCK, MPI\_WIN\_LOCK\_ALL: MPI\_MODE\_NOCHECK no other process holds, or will attempt to acquire, a conflicting 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.

Advice to users. Note that the nostore and noprecede flags provide information on what happened *before* the call; the noput and nosucceed flags provide information on what will happen *after* the call. (*End of advice to users.*)

#### 11.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments 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 communication.

# 11.6 Error Handling

#### 11.6.1 Error Handlers

Errors occurring during calls to 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 on page 340).

#### 11.6.2 Error Classes

The 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 or MPI\_ERR\_RANK.

 $\overline{7}$ 

 $45 \\ 46$ 

1	MPI_ERR_WIN	invalid win argument
2	 MPI_ERR_BASE	invalid base argument
3	 MPI_ERR_SIZE	invalid size argument
4	MPI_ERR_DISP	invalid disp argument
5	MPI_ERR_LOCKTYPE	invalid locktype argument
6	MPI_ERR_ASSERT	invalid assert argument
7	MPI_ERR_RMA_CONFLICT	conflicting accesses to window
8	MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls
9	MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case
10		of a window created with
11		MPI_WIN_CREATE_DYNAMIC, target memory is not
12		attached)
13	MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource
14		exhaustion)
15	MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the
16		group of the specified communicator cannot expose
17		shared memory)
18	MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called
19		function
20		

Table 11.2: Error classes in one-sided communication routines

# 11.7 Semantics and Correctness

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.

- An RMA operation is completed at the origin by the ensuing call to MPI\_WIN\_COMPLETE, MPI\_WIN\_FENCE, 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.
- 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.
- 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.
- 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.

- 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, 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.
- 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, 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 eventually becomes visible in the private copy in process memory without additional RMA calls.

14The MPI\_WIN\_FENCE or MPI\_WIN\_WAIT call that completes the transfer from public 15copy to private copy (6) is the same call that completes the put or accumulate operation in 16the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then 17 the update of the public window copy is complete as soon as the updating process executed 18 MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL. In the RMA separate memory model, the 19update of a private copy in the process memory may be delayed until the target process 20executes a synchronization call on that window (6). Thus, updates to process memory can 21always be delayed in the RMA separate memory model until the process executes a suitable 22synchronization call, while they must complete in the RMA unified model without additional 23synchronization calls. If fence or post-start-complete-wait synchronization is used, updates  $^{24}$ to a public window copy can be delayed in both memory models until the window owner 25executes a synchronization call. When passive target synchronization (lock/unlock or even 26flush) is used, it is necessary to update the public window copy in the RMA separate model, 27or the private window copy in the RMA unified model, even if the window owner does not 28 execute any related synchronization call. 29

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 certain 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 operations 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), 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 difficult to use MPI RMA to implement programming models—

1 2

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11

12

13

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

 $46 \\ 47$ 

	454	CHAPTER 11. ONE-SIDED COMMUNICATIONS
1 2 3 4 5 6		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. ( <i>End of rationale.</i> )
7 8 9 10 11		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. ( <i>End of advice to implementors.</i> )
12 13		A program with a well-defined outcome in the MPI_WIN_SEPARATE memory model obey the following rules.
14 15 16 17	1.	A location in a window must not be accessed 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.
18 19 20 21 22 23 24	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 window copy. 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.
25 26 27 28 29 30 31	3.	A put or accumulate must not access a target window once a load/store update or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.
32 33 34 35 36 37 38 39 40 41 42		<i>Rationale.</i> The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library would 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 considered prohibitive. ( <i>End of rationale.</i> )
42 43 44 45 46 47 48	the p and acces	Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize private and public window copies (that is, updates to one are made visible to the other). In the MPI_WIN_UNIFIED memory model, the rules are much simpler because the public private windows are the same. However, there are restrictions to avoid concurrent as to the same memory locations by different processes. The rules that a program with ll-defined outcome must obey in this case are:

- 1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.
- 2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, 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 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 or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

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. 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. 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 updates to memory with store operations without requiring an RMA 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 elsewhere 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 and until the update completes at the target. There is one exception to this rule: in the case where the same location 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 store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation 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 and 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.

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

 $\frac{44}{45}$ 

46

47

1	Advice to users. A user can write correct programs by following the following rules:
2	fence: During each period between fence calls, each window is either updated by put
$\frac{3}{4}$	or accumulate calls, or updated by stores, but not both. Locations updated by
4 5	put or accumulate calls should not be accessed during the same period (with
6	the exception of concurrent updates to the same location by accumulate calls).
7	Locations accessed by get calls should not be updated during the same period.
8	<b>post-start-complete-wait:</b> A window should not be updated with store operations
9	while posted if it is being updated by put or accumulate calls. Locations updated
10	by put or accumulate calls should not be accessed while the window is posted
11 12	(with the exception of concurrent updates to the same location by accumulate
12	calls). Locations accessed by get calls should not be updated while the window is posted.
14	With the post-start synchronization, the target process can tell the origin process
15	that its window is now ready for RMA access; with the complete-wait synchro-
16	nization, the origin process can tell the target process that it has finished its
17	RMA accesses to the window.
18	lock: Updates to the window are protected by exclusive locks if they may conflict.
19 20	Nonconflicting accesses (such as read-only accesses or accumulate accesses) are
21	protected by shared locks, both for load/store accesses and for RMA accesses.
22	${\bf changing \ window \ or \ synchronization \ mode: \ One \ can \ change \ synchronization \ mode,}$
23	or change the window used to access a location that belongs to two overlapping
24	windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI_WIN_FENCE, if
25	RMA accesses to the window are synchronized with fences; after a local call
26 27	to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-
28	wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK or
29	MPI_WIN_UNLOCK_ALL if the accesses are synchronized with locks.
30	In addition, a process should not access the local buffer of a get operation until the
31	operation is complete, and should not update the local buffer of a put or accumulate
32	operation until that operation is complete.
33 34	The RMA synchronization operations define when updates are guaranteed to become
35	visible in public and private windows. Updates may become visible earlier, but such
36	behavior is implementation dependent. (End of advice to users.)
37	
38	The semantics are illustrated by the following examples:
39	<b>Example 11.6</b> The following example demonstrates updating a memory location inside
40 41	a window for the separate memory model, according to Rule 5. The $MPI\_WIN\_LOCK$
42	and MPI_WIN_UNLOCK calls around the store to X in process B are necessary to ensure
43	consistency between the public and private copies of the window.
44	
45	
46	
47	

 $^{48}$ 

Process A:	Process B:	1
	window location X	2
		3
	MPI_Win_lock(EXCLUSIVE,B)	4
	store X /* local update to private copy of B */	5
	MPI_Win_unlock(B)	6
	<pre>/* now visible in public window copy */</pre>	7
	1 10	8
MPI_Barrier	MPI_Barrier	9
		10
MPI_Win_lock(EXCLUSIVE,B)		11
MPI_Get(X) /* ok, read fro	om public window */	12
MPI_Win_unlock(B)	-	13
		14
Exemple 11.7 In the DMA	wifed model although the public and private conics of the	15
-	inified model, although the public and private copies of the	16
	ition must be used when combining load/stores and multi-	17
	bugh the following example appears correct, the compiler or	18
	to X after the barrier, possibly resulting in the MPI_GET	19
returning an incorrect value of	λ.	20
Process A:	Process B:	21
TIOCESS A.	window location X	22
	WINDOW LOCADION A	23
	A at any $X$ (* undet at a private knublic conv. of $P$ *(	24

	<pre>store X /* update to private&amp;public copy of B */</pre>
MPI_Barrier	MPI_Barrier
MPI_Win_lock_all	
MPI_Get(X) /* ok, read from	n window */
MPI_Win_flush_local(B)	
/* read value in X */	
MPI_Win_unlock_all	

MPI\_BARRIER provides process synchronization, but not 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** 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.

0 0			43
			40
Process A:	Process B:		44
	window location	ı X	45
			46
MPI_Win_lock(EXCLUSIVE,B	)		47
<pre>MPI_Put(X) /* update to</pre>	public window */		48

1 2	MPI_Win_unlock(B)	
3	MPI_Barrier	MPI_Barrier
4 5		MPI_Win_lock(EXCLUSIVE,B)
6		/* now visible in private copy of B */
7		load X
8		MPI_Win_unlock(B)
9	Note that in this example the	e barrier is not critical to the semantic correctness. The
10 11		es a remote process will not modify the public copy after
12	0	the private and public copies. A polling implementation
13	looking for changes in X on proc	cess B would be semantically correct. The barrier is required
14	to ensure that process A perfor	ms the put operation before process B performs the load of
15	Х.	
16	Essential 11.0 Cinciler to Essen	
17	-	mple 11.7, the following example is unsafe even in the unified n not be guaranteed to occur after the MPI_BARRIER. While
18	,	plicitly synchronize the public and private copies through
19		PUT will update both the public and private copies of the
20		bad could result in old values of X being returned. Compiler
21 22		s could ensure the load occurs after the data is updated, or
23	explicit one-sided synchronizati	on calls can be used to ensure the proper result.
24		
25	Process A:	Process B:
26	MPI_Win_lock_all	window location X
27	MPI_Put(X) /* update to wi	ndow */
28	MPI_Win_flush(B)	
29		
30 31	MPI_Barrier	MPI_Barrier
32		load X
33	MPI_Win_unlock_all	
34		
35	<b>Example 11.10</b> The following	ng example further clarifies Rule 5. MPI_WIN_LOCK and
36	-	update the public copy of a window with changes to the
37	private copy. Therefore, there is	s no guarantee that process A in the following sequence will
38	see the value of ${\tt X}$ as updated b	y the local store by process B before the lock.
39		
40	Process A:	Process B:
41 42		window location X
43		store X /* update to private copy of B */
44		MPI_Win_lock(SHARED,B)
45	MPI_Barrier	MPI_Barrier
46		
47	MPI_Win_lock(SHARED,B)	
48	MPI_Get(X) /* X may be the	X before the store */

MPI_Win_unlock(B)			
	<pre>MPI_Win_unlock(B) /* update on X now visible in public window */</pre>		
The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would guarantee process A would see the updated value of X, as the public copy of the window would be explicitly synchronized with the private copy.			
<b>Example 11.11</b> Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. It is <i>not</i> guaranteed that process B reads the value of X as per the local update by process A, because neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in the public window copy.			
Process A: window location X window location Y	Process B:		
store Y MPI_Win_post(A,B) /* Y vis MPI_Win_start(A)	ible in public window */ MPI_Win_start(A)		
store X /* update to priva	te window */		
MPI_Win_complete MPI_Win_wait	MPI_Win_complete		
	visible in public window */		
MPI_Barrier	MPI_Barrier		
	<pre>MPI_Win_lock(EXCLUSIVE,A) MPI_Get(X) /* may return an obsolete value */ MPI_Get(Y) MPI_Win_unlock(A)</pre>		

To allow process B to read the value of X stored by A the local store must be replaced by a local MPI\_PUT that updates the public window copy. Note that by this replacement X may become visible in the private copy of process A only after the MPI\_WIN\_WAIT call in process A. The update to Y made before the MPI\_WIN\_POST call is visible in the public window after the MPI\_WIN\_POST call and therefore process B will read the proper value of Y. The MPI\_GET(Y) call could be moved to the epoch started by the MPI\_WIN\_START operation, and process B would still get the value stored by process A.

**Example 11.12** The following example demonstrates the interaction of general active target synchronization with local read operations with the RMA separate memory model. Rules 5 and 6 do *not* guarantee that the private copy of X at process B has been updated before the load takes place.

```
1
     Process A:
                                    Process B:
\mathbf{2}
                                     window location X
3
4
     MPI_Win_lock(EXCLUSIVE,B)
\mathbf{5}
     MPI_Put(X) /* update to public window */
6
     MPI_Win_unlock(B)
7
8
     MPI_Barrier
                                    MPI_Barrier
9
10
                                    MPI_Win_post(B)
11
                                    MPI_Win_start(B)
12
13
                                     load X /* access to private window */
14
                                            /* may return an obsolete value */
15
16
                                    MPI_Win_complete
17
                                    MPI_Win_wait
18
```

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.

# <sup>22</sup> 11.7.1 Atomicity

19

20 21

36

37

23The outcome of concurrent accumulate operations to the same location with the same  $^{24}$ predefined datatype is as if the accumulates were done at that location in some serial 25order. Additional restrictions on the operation apply; see the info key accumulate\_ops in 26Section 11.2.1. Concurrent accumulate operations with different origin and target pairs are 27not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is 28executed atomically. The effect of this lack of atomicity is limited: The previous correctness 29 conditions imply that a location updated by a call to an accumulate operation cannot be 30 accessed by a load or an RMA call other than accumulate until the accumulate operation has  $^{31}$ completed (at the target). Different interleavings can lead to different results only to the 32 extent that computer arithmetics are not truly associative or commutative. The outcome 33 of accumulate operations with overlapping types of different sizes or target displacements 34is undefined. 35

# 11.7.2 Ordering

38Accumulate calls enable element-wise atomic read and write to remote memory locations. 39 MPI specifies ordering between accumulate operations from one process to the same (or 40overlapping) memory locations at another process on a per-datatype granularity. The de-41 fault ordering is strict ordering, which guarantees that overlapping updates from the same 42source to a remote location are committed in program order and that reads (e.g., with 43MPI\_GET\_ACCUMULATE) and writes (e.g., with MPI\_ACCUMULATE) are executed and 44committed in program order. Ordering only applies to operations originating at the same 45origin that access overlapping target memory regions. MPI does not provide any guarantees 46for accesses or updates from different origins to overlapping target memory regions.

<sup>47</sup> The default strict ordering may incur a significant performance penalty. MPI specifies
 <sup>48</sup> the info key accumulate\_ordering to allow relaxation of the ordering semantics when specified

to any window creation function. The values for this key are as follows. If set to none, then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in MPI-2 but is *not* the default in MPI-3. The key can be set to a comma-separated list of required access orderings at the target. Allowed values in the comma-separated list are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-after-write 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. 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 readafter-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 put and get. Put and get 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 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 442. 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 447. 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.

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

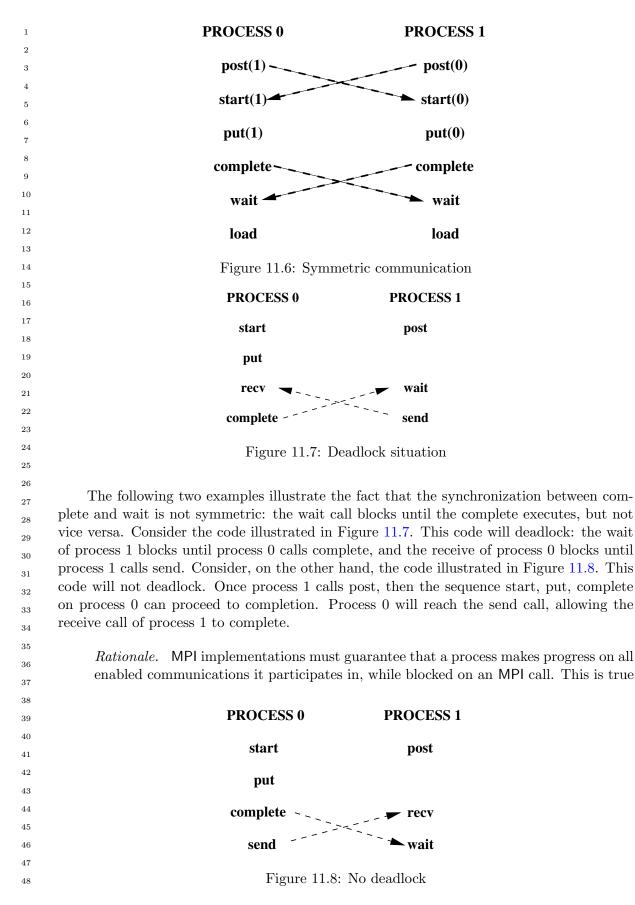
43

44

45

46

47



for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 11.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is 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, 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 call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI Forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but 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 values 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:

3
3
3
5
2
4

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 17.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran <sup>47</sup> compilers will avoid this problem, without disabling compiler optimizations. However, in <sup>48</sup>

1

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

43 44

 $45 \\ 46$ 

order to avoid register coherence problems in a completely portable manner, users should
 restrict their use of RMA windows to variables stored in modules or

COMMON blocks. To prevent problems with the argument copying and register optimization
 done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. Sections
 Solutions to The (Poorly Performing) Fortran VOLATILE Attribute on pages 638–643
 discuss several solutions for the problem in this example.

# 11.8 Examples

**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.

```
15
     . . .
16
     while(!converged(A)){
17
       update(A);
18
       MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
19
       for(i=0; i < toneighbors; i++)</pre>
20
         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
21
                                 todisp[i], 1, totype[i], win);
22
       MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
23
       }
```

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 (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 11.14 Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

```
. . .
35
     while(!converged(A)){
36
       update_boundary(A);
37
       MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
38
       for(i=0; i < fromneighbors; i++)</pre>
39
         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
40
                           fromdisp[i], 1, fromtype[i], win);
41
       update_core(A);
42
       MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
43
       }
44
```

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

8

9 10 11

12

13

14

 $^{24}$ 

29

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.

```
. . .
                                                                                     38
if (!converged(A0,A1))
                                                                                     39
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
                                                                                     40
MPI_Barrier(comm0);
                                                                                     41
/* the barrier is needed because the start call inside the
                                                                                     42
loop uses the nocheck option */
                                                                                     43
while(!converged(A0, A1)){
                                                                                     44
  /* communication on AO and computation on A1 */
                                                                                     45
  update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
                                                                                     46
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
                                                                                     47
  for(i=0; i < fromneighbors; i++)</pre>
                                                                                     48
```

 $\mathbf{2}$ 

```
1
         MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
\mathbf{2}
                     fromdisp0[i], 1, fromtype0[i], win0);
3
       update1(A1); /* local update of A1 that is
4
                        concurrent with communication that updates AO */
5
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
6
       MPI_Win_complete(win0);
\overline{7}
       MPI_Win_wait(win0);
8
9
       /* communication on A1 and computation on A0 */
10
       update2(A0, A1); /* local update of A0 that depends on A1 (and A0) */
11
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
12
       for(i=0; i < fromneighbors; i++)</pre>
13
         MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
14
                      fromdisp1[i], 1, fromtype1[i], win1);
15
       update1(A0); /* local update of A0 that depends on A0 only,
16
                       concurrent with communication that updates A1 */
17
       if (!converged(A0,A1))
18
         MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
19
       MPI_Win_complete(win1);
       MPI_Win_wait(win1);
20
21
       }
22
```

A process posts the local window associated with win0 before it completes RMA accesses to the remote windows associated with win1. When the wait(win1) call returns, then all neighbors of the calling process have posted the windows associated with win0. Conversely, when the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win1. Therefore, the nocheck option can be used with the calls to MPI\_WIN\_START.

Put calls can be used, instead of get calls, if the area of array A0 (resp. A1) used by the update(A1, A0) (resp. update(A0, A1)) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

In the next several examples, for conciseness, the expression

z = MPI\_Get\_accumulate(...)

means to perform an MPI\_GET\_ACCUMULATE with the result buffer (given by result\_addr
 in the description of MPI\_GET\_ACCUMULATE) on the left side of the assignment, in this
 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.

46

33 34

35

- 47
- 48

Process A:	Process B:	1
MPI_Win_lock_all	MPI_Win_lock_all	2
window location X		3
X=2		4
MPI_Win_sync		5
MPI_Barrier	MPI_Barrier	6
		7
MPI_Accumulate(X, MPI_SUM, -1)	MPI_Accumulate(X, MPI_SUM, -1)	8
		9
stack variable z	stack variable z	10
do	do	11
<pre>z = MPI_Get_accumulate(X,</pre>	<pre>z = MPI_Get_accumulate(X,</pre>	12
MPI_NO_OP, 0)	MPI_NO_OP, 0)	13
MPI_Win_flush(A)	MPI_Win_flush(A)	14
while(z!=0)	while(z!=0)	15
		16
MPI_Win_unlock_all	MPI_Win_unlock_all	17
		18

**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.

<pre>window location X window location Y 27 window location T 28 PMPI_Win_lock_all MPI_Win_lock_all 30 X=1 Y=1 31 MPI_Win_sync MPI_Win_sync 32 MPI_Barrier MPI_Barrier 33 MPI_Accumulate(T, MPI_REPLACE, 1) MPI_Accumulate(T, MPI_REPLACE, 0) 34 stack variables t,y stack variable t,x 35 t=1 t=0 36 y=MPI_Get_accumulate(Y, x=MPI_Get_accumulate(X, 37 MPI_NO_OP, 0) MPI_NO_OP, 0) 38 while(y==1 &amp;&amp; t==1) do while(x==1 &amp;&amp; t==0) do 39 y=MPI_Get_accumulate(Y, x=MPI_Get_accumulate(X, 40 MPI_NO_OP, 0) MPI_NO_OP, 0) 41 t=MPI_Get_accumulate(T, t=MPI_Get_accumulate(T, 42 MPI_NO_OP, 0) MPI_NO_OP, 0) 41 t=MPI_Get_accumulate(T, t=MPI_Get_accumulate(T, 42 MPI_NO_OP, 0) MPI_NO_OP, 0) 41 t=MPI_OP, 0) MPI_NO_OP, 0) 41 </pre>
29         MPI_Win_lock_all       MPI_Win_lock_all       30         X=1       Y=1       31         MPI_Win_sync       MPI_Win_sync       32         MPI_Barrier       MPI_Barrier       33         MPI_Accumulate(T, MPI_REPLACE, 1)       MPI_Accumulate(T, MPI_REPLACE, 0)       34         stack variables t,y       stack variable t,x       35         t=1       t=0       36         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       37         MPI_NO_OP, 0)       MPI_NO_OP, 0)       38         while(y==1 && t==1) do       while(x==1 && t==0) do       39         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_Get_accumulate(T,       t=MPI_Get_accumulate(T,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_OOP, 0)       MPI_NO_OP, 0)       41
MPI_Win_lock_all       MPI_Win_lock_all       30         X=1       Y=1       31         MPI_Win_sync       MPI_Win_sync       32         MPI_Barrier       MPI_Barrier       33         MPI_Accumulate(T, MPI_REPLACE, 1)       MPI_Accumulate(T, MPI_REPLACE, 0)       34         stack variables t,y       stack variable t,x       35         t=1       t=0       36         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       37         MPI_NO_OP, 0)       MPI_NO_OP, 0)       38         while(y==1 && t=1) do       while(x==1 && t=0) do       39         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_Get_accumulate(T,       t=MPI_Get_accumulate(T,       42         MPI_NO_OP, 0)       MPI_NO_OP, 0)       43
NFI_WIN_IOCK_allNFI_WIN_IOCK_allX=1Y=131MPI_Win_syncMPI_Win_sync32MPI_BarrierMPI_Barrier33MPI_Accumulate(T, MPI_REPLACE, 1)MPI_Accumulate(T, MPI_REPLACE, 0)34stack variables t,ystack variable t,x35t=1t=036y=MPI_Get_accumulate(Y,x=MPI_Get_accumulate(X,37MPI_NO_OP, 0)MPI_NO_OP, 0)38while(y=1 && t=1) dowhile(x=1 && t=0) do39y=MPI_Get_accumulate(Y,x=MPI_Get_accumulate(X,40MPI_NO_OP, 0)MPI_NO_OP, 0)41t=MPI_Get_accumulate(T,t=MPI_Get_accumulate(T,42MPI_NO_OP, 0)MPI_NO_OP, 0)43
A-1I-1MPI_Win_syncMPI_Win_sync32MPI_BarrierMPI_Barrier33MPI_Accumulate(T, MPI_REPLACE, 1)MPI_Accumulate(T, MPI_REPLACE, 0)34stack variables t,ystack variable t,x35t=1t=036y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)x=MPI_Get_accumulate(X, MPI_NO_OP, 0)38while(y=1 && t=1) dowhile(x=1 && t=0) do39y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)x=MPI_Get_accumulate(X, MPI_NO_OP, 0)41t=MPI_Get_accumulate(T, MPI_NO_OP, 0)t=MPI_Get_accumulate(T, MPI_NO_OP, 0)43
MP1_wIII_syncMP1_wIII_syncMPI_BarrierMPI_Barrier33MPI_Accumulate(T, MPI_REPLACE, 1)MPI_Accumulate(T, MPI_REPLACE, 0)34stack variables t,ystack variable t,x35t=1t=036y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)x=MPI_Get_accumulate(X,37MPI_NO_OP, 0)MPI_NO_OP, 0)38while(y==1 && t==1) do y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)x=MPI_Get_accumulate(X,40MPI_NO_OP, 0)MPI_NO_OP, 0)41t=MPI_Get_accumulate(T, MPI_NO_OP, 0)t=MPI_Get_accumulate(T, MPI_NO_OP, 0)43
MPI_BarrierMPI_BarrierMPI_Accumulate(T, MPI_REPLACE, 1)MPI_Accumulate(T, MPI_REPLACE, 0)34stack variables t,ystack variable t,x35t=1t=036y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)x=MPI_Get_accumulate(X,37while(y==1 && t==1) do y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)while(x==1 && t==0) do39while(g==1 && t==1) do y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)x=MPI_Get_accumulate(X,40MPI_NO_OP, 0)MPI_NO_OP, 0)41t=MPI_Get_accumulate(T, MPI_NO_OP, 0)t=MPI_Get_accumulate(T, MPI_NO_OP, 0)43
MP1_Accumulate(1, MP1_REPLACE, 1)       MP1_Accumulate(1, MP1_REPLACE, 0)         stack variables t,y       stack variable t,x       35         t=1       t=0       36         y=MP1_Get_accumulate(Y,       x=MP1_Get_accumulate(X,       37         MP1_NO_OP, 0)       MP1_NO_OP, 0)       38         while(y==1 && t==1) do       while(x==1 && t==0) do       39         y=MP1_Get_accumulate(Y,       x=MP1_Get_accumulate(X,       40         MP1_NO_OP, 0)       MP1_NO_OP, 0)       41         t=MP1_Get_accumulate(T,       t=MP1_Get_accumulate(T,       42         MP1_NO_OP, 0)       MP1_NO_OP, 0)       43
stack valiables t,y       stack valiable t,x       accumulate t = 0       36         t=1       t=0       36         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       37         MPI_NO_OP, 0)       MPI_NO_OP, 0)       38         while(y==1 && t==1) do       while(x==1 && t==0) do       39         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_Get_accumulate(T,       t=MPI_Get_accumulate(T,       42         MPI_NO_OP, 0)       MPI_NO_OP, 0)       43
t-1       t-0       t-0         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       37         MPI_NO_OP, 0)       MPI_NO_OP, 0)       38         while(y==1 && t==1) do       while(x==1 && t==0) do       39         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_Get_accumulate(T,       t=MPI_Get_accumulate(T,       42         MPI_NO_OP, 0)       MPI_NO_OP, 0)       43
y=MF1_Get_accumulate(T,       x=MF1_Get_accumulate(X,         MPI_NO_OP, 0)       MPI_NO_OP, 0)       38         while(y==1 && t==1) do       while(x==1 && t==0) do       39         y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_Get_accumulate(T,       t=MPI_Get_accumulate(T,       42         MPI_NO_OP, 0)       MPI_NO_OP, 0)       43
<pre>while(y==1 &amp;&amp; t==1) do</pre>
y=MPI_Get_accumulate(Y,       x=MPI_Get_accumulate(X,       40         MPI_NO_OP, 0)       MPI_NO_OP, 0)       41         t=MPI_Get_accumulate(T,       t=MPI_Get_accumulate(T,       42         MPI_NO_OP, 0)       MPI_NO_OP, 0)       43
y=MP1_Get_accumulate(1,x=MP1_Get_accumulate(x,MP1_NO_OP, 0)MP1_NO_OP, 0)41t=MP1_Get_accumulate(T,t=MP1_Get_accumulate(T,42MP1_NO_OP, 0)MP1_NO_OP, 0)43
MP1_N0_0P, 0)       MP1_N0_0P, 0)       42         t=MPI_N0_0P, 0)       MP1_N0_0P, 0)       43
t-MP1_Get_accumulate(1,t-MP1_Get_accumulate(1,MP1_NO_OP, 0)MP1_NO_OP, 0)
MPI_Win_flush_all MPI_Win_flush(A) 44
done done <sup>45</sup>
<pre>// critical region // critical region 46</pre>
MPI_Accumulate(X, MPI_REPLACE, 0) MPI_Accumulate(Y, MPI_REPLACE, 0) 47
MPI_Win_unlock_all MPI_Win_unlock_all 48

 $^{24}$ 

Example 11.20 Implementing a critical region between multiple processes with compare
 and swap. The call to MPI\_WIN\_SYNC is necessary on Process A after local initialization
 of A to guarantee the public copy has been updated with the initialization value found in
 the private copy. It would also be valid to call MPI\_ACCUMULATE with MPI\_REPLACE to
 directly initialize the public copy. A call to MPI\_WIN\_FLUSH would be necessary to assure
 A in the public copy of Process A had been updated before the barrier.

```
7
                                                 Process B...:
     Process A:
8
     MPI_Win_lock_all
                                                 MPI_Win_lock_all
9
     atomic location A
10
     A=0
11
     MPI_Win_sync
12
     MPI_Barrier
                                                 MPI_Barrier
13
     stack variable r=1
                                                 stack variable r=1
14
     while(r != 0) do
                                                 while(r != 0) do
15
       r = MPI_Compare_and_swap(A, 0, 1)
                                                   r = MPI_Compare_and_swap(A, 0, 1)
16
       MPI_Win_flush(A)
                                                   MPI_Win_flush(A)
17
     done
                                                 done
18
     // critical region
                                                 // critical region
19
     r = MPI_Compare_and_swap(A, 1, 0)
                                                 r = MPI_Compare_and_swap(A, 1, 0)
20
                                                 MPI_Win_unlock_all
     MPI_Win_unlock_all
21
22
23
     Example 11.21 The following example shows how request-based operations can be used
^{24}
     to overlap communication with computation. Each process fetches, processes, and writes
25
     the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to
26
     allow up to M communication operations to overlap with computation.
27
28
     int
                  i, j;
```

```
^{29}
     MPI_Win
                  win;
30
     MPI_Request put_req[M] = { MPI_REQUEST_NULL };
31
     MPI_Request get_req;
32
                  **baseptr;
     double
33
     double
                  data[M][N];
34
35
     MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
36
       MPI_COMM_WORLD, baseptr, &win);
37
38
     MPI_Win_lock_all(0, win);
39
40
     for (i = 0; i < NSTEPS; i++) {</pre>
^{41}
      if (i<M)
42
        j=i;
43
      else
44
        MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
45
46
      MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
47
                &get_req);
48
      MPI_Wait(&get_req,MPI_STATUS_IGNORE);
```

```
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.

```
. . .
                                                                                       18
#define NUM_ELEMS 10
                                                                                       19
                                                                                      20
/* Linked list pointer */
                                                                                      21
typedef struct {
                                                                                      22
  MPI_Aint disp;
                                                                                      23
  int
            rank;
                                                                                       ^{24}
} llist_ptr_t;
                                                                                       25
                                                                                       26
/* Linked list element */
                                                                                      27
typedef struct {
                                                                                      28
  llist_ptr_t next;
                                                                                      29
  int value;
                                                                                       30
} llist_elem_t;
                                                                                       31
                                                                                       32
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
                                                                                      33
                                                                                      34
/* List of locally allocated list elements. */
                                                                                      35
static llist_elem_t **my_elems = NULL;
                                                                                      36
static int my_elems_size = 0;
                                                                                      37
static int my_elems_count = 0;
                                                                                       38
                                                                                       39
/* Allocate a new shared linked list element */
                                                                                       40
MPI_Aint alloc_elem(int value, MPI_Win win) {
                                                                                       41
  MPI_Aint disp;
                                                                                      42
  llist_elem_t *elem_ptr;
                                                                                      43
                                                                                      44
  /* Allocate the new element and register it with the window */
                                                                                       45
  MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
                                                                                       46
  elem_ptr->value = value;
                                                                                       47
  elem_ptr->next = nil;
                                                                                       48
```

469

1

 $\mathbf{2}$ 

3

4 5 6

7

8 9

10

11

12

13

14

15

```
1
       MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
2
3
       /* Add the element to the list of local elements so we can free
4
          it later. */
5
       if (my_elems_size == my_elems_count) {
6
         my_elems_size += 100;
7
         my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
8
       }
9
       my_elems[my_elems_count] = elem_ptr;
10
       my_elems_count++;
11
12
       MPI_Get_address(elem_ptr, &disp);
13
       return disp;
14
     }
15
16
     int main(int argc, char *argv[]) {
17
       int
                      procid, nproc, i;
^{18}
                      llist_win;
       MPI_Win
19
       llist_ptr_t
                     head_ptr, tail_ptr;
20
21
       MPI_Init(&argc, &argv);
22
23
       MPI_Comm_rank(MPI_COMM_WORLD, &procid);
^{24}
       MPI_Comm_size(MPI_COMM_WORLD, &nproc);
25
26
       MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
27
28
       /* Process 0 creates the head node */
29
       if (procid == 0)
30
         head_ptr.disp = alloc_elem(-1, llist_win);
^{31}
32
       /* Broadcast the head pointer to everyone */
33
       head_ptr.rank = 0;
34
       MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
35
       tail_ptr = head_ptr;
36
37
       /* Lock the window for shared access to all targets */
38
       MPI_Win_lock_all(0, llist_win);
39
40
       /* All processes concurrently append NUM_ELEMS elements to the list */
41
       for (i = 0; i < NUM_ELEMS; i++) {</pre>
42
         llist_ptr_t new_elem_ptr;
43
         int success;
44
45
         /* Create a new list element and attach it to the window */
46
         new_elem_ptr.rank = procid;
47
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
48
```

}

. . .

```
1
  /* Append the new node to the list. This might take multiple
                                                                                  2
     attempts if others have already appended and our tail pointer
                                                                                  3
     is stale. */
                                                                                  4
  do {
    llist_ptr_t next_tail_ptr = nil;
                                                                                  5
                                                                                  6
                                                                                  7
    MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
         (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
                                                                                  8
         (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.rank),
                                                                                  9
                                                                                  10
        llist_win);
                                                                                 11
    MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                 12
    success = (next_tail_ptr.rank == nil.rank);
                                                                                 13
                                                                                 14
                                                                                 15
    if (success) {
      MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
                                                                                 16
                                                                                  17
          (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.disp), 1,
                                                                                 18
          MPI_AINT, MPI_REPLACE, llist_win);
                                                                                 19
      MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                 20
                                                                                 21
      tail_ptr = new_elem_ptr;
                                                                                 22
                                                                                 23
    } else {
                                                                                 24
      /* Tail pointer is stale, fetch the displacement. May take
                                                                                 25
         multiple tries if it is being updated. */
                                                                                 26
      do {
        MPI_Get_accumulate( NULL, 0, MPI_AINT, &next_tail_ptr.disp,
                                                                                 27
            1, MPI_AINT, tail_ptr.rank,
                                                                                 28
             (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.disp),
                                                                                 29
                                                                                 30
            1, MPI_AINT, MPI_NO_OP, llist_win);
                                                                                 31
        MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                 32
                                                                                 33
      } while (next_tail_ptr.disp == nil.disp);
                                                                                 34
      tail_ptr = next_tail_ptr;
                                                                                 35
    }
                                                                                 36
  } while (!success);
                                                                                 37
                                                                                 38
                                                                                 39
MPI_Win_unlock_all(llist_win);
MPI_Barrier( MPI_COMM_WORLD );
                                                                                  40
                                                                                 41
                                                                                 42
/* Free all the elements in the list */
for ( ; my_elems_count > 0; my_elems_count--) {
                                                                                 43
  MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
                                                                                 44
  MPI_Free_mem(my_elems[my_elems_count-1]);
                                                                                 45
}
                                                                                  46
                                                                                  47
MPI_Win_free(&llist_win);
                                                                                  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. These calls can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. 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 to 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 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.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the ap-

```
1
     plication. For a generalized request, the operation associated with the request is per-
\mathbf{2}
     formed by the application; therefore, the application must notify MPI through a call to
3
     MPI_GREQUEST_COMPLETE when the operation completes. MPI maintains the "comple-
4
     tion" status of generalized requests. Any other request state has to be maintained by the
5
     user.
6
          A new generalized request is started with
7
8
     MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
9
10
       IN
                 query_fn
                                              callback function invoked when request status is queried
11
                                              (function)
12
       IN
                 free_fn
                                              callback function invoked when request is freed (func-
13
                                              tion)
14
                 cancel_fn
       IN
                                             callback function invoked when request is cancelled
15
                                              (function)
16
17
       IN
                 extra_state
                                              extra state
18
       OUT
                 request
                                              generalized request (handle)
19
20
     int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
21
                     MPI_Grequest_free_function *free_fn,
22
                     MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
23
                     MPI_Request *request)
^{24}
25
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
26
                     ierror) BIND(C)
27
          PROCEDURE(MPI_Grequest_query_function) :: query_fn
28
          PROCEDURE(MPI_Grequest_free_function) :: free_fn
29
          PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
30
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
31
          TYPE(MPI_Request), INTENT(OUT) :: request
32
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
33
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
34
                     IERROR)
35
          INTEGER REQUEST, IERROR
36
          EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
37
          INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
38
39
40
           Advice to users.
                              Note that a generalized request is of the same type as regular
41
           requests, in C and Fortran. (End of advice to users.)
42
          The call starts a generalized request and returns a handle to it in request.
43
         The syntax and meaning of the callback functions are listed below. All callback func-
44
     tions are passed the extra_state argument that was associated with the request by the
45
46
     starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined
47
     state for the request.
         In C, the query function is
48
```

CHAPTER 12. EXTERNAL INTERFACES

tone left int NDT decouvert more for this (as it to the	1
<pre>typedef int MPI_Grequest_query_function(void *extra_state,</pre>	2
MPI_Status *status);	3
in Fortran with the mpi_f08 module	
ABSTRACT INTERFACE	4
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)	5
BIND(C)	6
TYPE(MPI_Status) :: status	7
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	8
INTEGER :: ierror	9
	10
in Fortran with the mpi module and the deprecated mpif.h	$^{11}$ ticketWG.
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)	12
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	13
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	14
	15
The query_fn function computes the status that should be returned for the generalized	16
request. The status also includes information about successful/unsuccessful cancellation of	17
the request (result to be returned by MPI_TEST_CANCELLED).	18
The query_fn callback is invoked by the MPI_{WAIT TEST}{ANY SOME ALL} call that	19
completed the generalized request associated with this callback. The callback function is	20
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when	21
the call occurs. In both cases, the callback is passed a reference to the corresponding	22
status variable passed by the user to the MPI call; the status set by the callback function	23
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or	24
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI	25
will pass a valid status object to query_fn, and this status will be ignored upon return of the	26
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE	27
is called on the request; it may be invoked several times for the same generalized request,	28
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also	29
that a call to $MPI_{WAIT TEST}_{SOME ALL}$ may cause multiple invocations of $query_{fn}$	30
callback functions, one for each generalized request that is completed by the MPI call. The	31
order of these invocations is not specified by MPI.	32
In C, the free function is	33
<pre>typedef int MPI_Grequest_free_function(void *extra_state);</pre>	34
in Fortran with the mpi_f08 module	35
ABSTRACT INTERFACE	36
SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)	37
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	38
INTEGER (KIND-FFI_ADDRESS_KIND) extra_state	39
TALEADY TOLIOI	40
in Fortran with the mpi module and the deprecated mpif.h	$_{41}$ ticketWG.
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)	42
INTEGER IERROR	43
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	44
	45
The free_fn function is invoked to clean up user-allocated resources when the generalized	46
request is freed.	47
	48

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 by MPI.

6 The free\_fn callback is also invoked for generalized requests that are freed by a call 7to MPI\_REQUEST\_FREE (no call to MPI\_{WAIT|TEST}{ANY|SOME|ALL} will occur for 8 such a request). In this case, the callback function will be called either in the MPI call 9 MPI\_REQUEST\_FREE(request), or in the MPI call MPI\_GREQUEST\_COMPLETE(request), 10 whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 11calls MPI\_REQUEST\_FREE and MPI\_GREQUEST\_COMPLETE have occurred. The request 12is not deallocated until after free\_fn completes. Note that free\_fn will be invoked only once 13per request by a correct program.

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 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 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 avoid accessing this stale handle. This is a special case in which MPI defers deallocating the object until a later time that is known by the user. (*End of advice to users.*)

In C, the cancel function is

typedef int MPI\_Grequest\_cancel\_function(void \*extra\_state, int complete);

<sup>28</sup> in Fortran with the  $mpi_f08$  module

<sup>29</sup> ABSTRACT INTERFACE

<sup>30</sup> SUBROUTINE MPI\_Grequest\_cancel\_function(extra\_state, complete, ierror)
 <sup>31</sup> BIND(C)

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

LOGICAL :: complete

INTEGER :: ierror

ticket WG.  $^{35}_{36}$ 

in Fortran with the mpi module and the deprecated mpif.h
 SUBROUTINE GREQUEST\_CANCEL\_FUNCTION(EXTRA\_STATE, COMPLETE, IERROR)
 INTEGER IERROR
 INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE
 LOGICAL COMPLETE

<sup>41</sup> The cancel\_fn function is invoked to start the cancelation of a generalized request. <sup>42</sup> It is called by MPI\_CANCEL(request). MPI passes complete=true to the callback function <sup>43</sup> if MPI\_GREQUEST\_COMPLETE was already called on the request, and

<sup>44</sup> complete=false otherwise.

<sup>45</sup> All callback functions return an error code. The code is passed back and dealt with as <sup>46</sup> appropriate for the error code by the MPI function that invoked the callback function. For <sup>47</sup> example, if error codes are returned then the error code returned by the callback function <sup>48</sup> will be returned by the MPI function that invoked the callback function. In the case of

14

15

16

17

18

19

20

21

22

23

24 25

26

27

32

33

34

an MPI\_{WAIT|TEST}{ANY} call that invokes both query\_fn and free\_fn, the MPI call will return the error code returned by the last callback, namely free\_fn. If one or more of the requests in a call to MPI\_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI\_ERR\_IN\_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free\_fn callback function. However, if the MPI function was passed MPI\_STATUSES\_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query\_fn must not set the error field of status since query\_fn may be called by MPI\_WAIT or MPI\_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query\_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (End of advice to users.)

MPI\_GREQUEST\_COMPLETE(request)

```
INOUT request generalized request (handle)
int MPI_Grequest_complete(MPI_Request request)
MPI_Grequest_complete(request, ierror) BIND(C)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
```

INTEGER REQUEST, IERROR

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, 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 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.*)

 $^{24}$ 

 $^{31}$ 

```
12.2.1 Examples
```

**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.

```
7
     typedef struct {
8
        MPI_Comm comm;
9
        int tag;
10
        int root;
^{11}
        int valin;
12
        int *valout;
13
        MPI_Request request;
14
        } ARGS;
15
16
17
     int myreduce(MPI_Comm comm, int tag, int root,
18
                    int valin, int *valout, MPI_Request *request)
19
     {
20
        ARGS *args;
21
        pthread_t thread;
22
23
        /* start request */
^{24}
        MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
25
26
        args = (ARGS*)malloc(sizeof(ARGS));
27
        args->comm = comm;
28
        args->tag = tag;
29
        args->root = root;
30
        args->valin = valin;
^{31}
        args->valout = valout;
32
        args->request = *request;
33
34
        /* spawn thread to handle request */
35
        /* The availability of the pthread_create call is system dependent */
36
        pthread_create(&thread, NULL, reduce_thread, args);
37
38
        return MPI_SUCCESS;
39
     }
40
^{41}
     /* thread code */
42
     void* reduce_thread(void *ptr)
43
     ſ
44
        int lchild, rchild, parent, lval, rval, val;
45
        MPI_Request req[2];
46
        ARGS *args;
47
48
        args = (ARGS*)ptr;
```

1

2 3

4

5

```
2
   /* compute left and right child and parent in tree; set
                                                                                    3
      to MPI_PROC_NULL if does not exist */
   /* code not shown */
                                                                                    4
                                                                                    5
   . . .
                                                                                    6
                                                                                    7
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
                                                                                    8
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    9
                                                                                    10
   val = lval + args->valin + rval;
                                                                                   11
   MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
                                                                                   12
   MPI_Grequest_complete((args->request));
                                                                                   13
                                                                                   14
   free(ptr);
                                                                                    15
   return(NULL);
}
                                                                                    16
                                                                                    17
                                                                                   18
int query_fn(void *extra_state, MPI_Status *status)
                                                                                   19
{
   /* always send just one int */
                                                                                   20
                                                                                   21
   MPI_Status_set_elements(status, MPI_INT, 1);
   /* can never cancel so always true */
                                                                                   22
                                                                                   23
   MPI_Status_set_cancelled(status, 0);
                                                                                   24
   /* choose not to return a value for this */
                                                                                   25
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                   26
   /* tag has no meaning for this generalized request */
   status->MPI_TAG = MPI_UNDEFINED;
                                                                                   27
   /* this generalized request never fails */
                                                                                   28
                                                                                   29
   return MPI_SUCCESS;
                                                                                   30
}
                                                                                   31
                                                                                   32
                                                                                   33
int free_fn(void *extra_state)
                                                                                   34
{
   /* this generalized request does not need to do any freeing */
                                                                                   35
   /* as a result it never fails here */
                                                                                   36
                                                                                   37
   return MPI_SUCCESS;
}
                                                                                   38
                                                                                   39
                                                                                    40
                                                                                   41
int cancel_fn(void *extra_state, int complete)
                                                                                   42
{
   /* This generalized request does not support cancelling.
                                                                                   43
                                                                                   44
      Abort if not already done. If done then treat as if cancel failed.*/
                                                                                   45
   if (!complete) {
                                                                                   46
     fprintf(stderr,
                                                                                   47
             "Cannot cancel generalized request - aborting program\n");
                                                                                   48
     MPI_Abort(MPI_COMM_WORLD, 99);
```

```
}
return MPI_SUCCESS;
}
```

#### 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 to use the same request 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 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 the 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:

```
25
26
```

MPI\_STATUS\_SET\_ELEMENTS(status, datatype, count)

```
27
       INOUT
                 status
                                            status with which to associate count (Status)
28
       IN
                 datatype
                                             datatype associated with count (handle)
29
30
       IN
                 count
                                            number of elements to associate with status (integer)
31
32
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
33
                    int count)
34
     MPI_Status_set_elements(status, datatype, count, ierror) BIND(C)
35
          TYPE(MPI_Status), INTENT(INOUT) :: status
36
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
37
          INTEGER, INTENT(IN) :: count
38
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
39
40
     MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
41
          INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
42
43
44
45
46
47
48
```

1

 $\mathbf{2}$ 

3

4 5

> 6 7

8

9

10

11

MPI STAT	US_SET_ELEMENTS_X(stat	us. datatype. count)	1
- INOUT	status	status with which to associate count (Status)	2
IN	datatype	datatype associated with count (handle)	3
IN	count	÷-	4 5
IIN	count	number of elements to associate with status (integer)	6
int MPT 9	Status set elements x(MPT	_Status *status, MPI_Datatype datatype,	7
1110 III 1_K	MPI_Count count)	_Dealas #Status, in 1_Datatype datatype,	8
NDT OL I			9
	MPI_Status), INTENT(INOU	datatype, count, ierror) BIND(C) T) :: status	10
	(MPI_Datatype), INTENT(IN		11 12
	GER(KIND = MPI_COUNT_KIND	· · ·	13
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	14
MPT STATI	JS SET ELEMENTS X(STATUS.	DATATYPE, COUNT, IERROR)	15
	GER STATUS(MPI_STATUS_SIZ		16
INTEC	GER (KIND=MPI_COUNT_KIND)	COUNT	17
These	functions modify the opage	a part of status so that a call to	18 19
	v 1 1	EMENTS_X will return count. MPI_GET_COUNT	20
	a compatible value.		21
			22
		ents is set instead of the count because the former	23
can	uear with a nonintegral numb	er of datatypes. (End of rationale.)	24
A sub	osequent call to MPI_GET_C	OUNT(status, datatype, count),	25 26
	_ELEMENTS(status, datatype,		20 27
MPI_GET_ELEMENTS_X(status, datatype, count) must use a datatype argument that has			28
	the same type signature as the datatype argument that was used in the call to MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X.		
MPI_STAI	US_SET_ELEMENTS OF MIP	_STATUS_SET_ELEMENTS_A.	30
Rati	onale. The requirement of	matching type signatures for these calls is similar	31
		a count is set by a receive operation: in that case,	32
	,	PI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X	33 34
		e signature as the datatype used in the receive call.	35
(Enc	l of rationale.)		36
			37
			38
	US_SET_CANCELLED(status	_,	39
INOUT	status	status with which to associate cancel flag (Status)	40 41
IN	flag	if true indicates request was cancelled (logical)	41
			43
int MPI_S	Status_set_cancelled(MPI_	Status *status, int flag)	44
MPI_Statu	<pre>us_set_cancelled(status,</pre>	flag, ierror) BIND(C)	45
	(MPI_Status), INTENT(INOU	T) :: status	46
	CAL, INTENT(OUT) :: flag		47
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	48

# MPI\_STATUS\_SET\_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR LOGICAL FLAG

If flag is set to true then a subsequent call to  $MPI\_TEST\_CANCELLED(status, flag)$  will also return flag = true, otherwise it will return false.

Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI\_GET\_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra\_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI\_RECV, may lead to unpredictable results and is strongly discouraged. (*End of advice to users.*)

12.4 MPI and Threads

<sup>18</sup> This section specifies the interaction between MPI calls and threads. The section lists mini-<sup>20</sup> mal requirements for *thread compliant* MPI implementations and defines functions that can <sup>21</sup> be used for initializing the thread environment. MPI may be implemented in environments <sup>22</sup> where threads are not supported or perform poorly. Therefore, MPI implementations are <sup>23</sup> not required to be thread compliant as defined in this section.

This section generally assumes a thread package similar to POSIX threads [39], 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 multi threaded. 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 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.
- <sup>46</sup> 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.

4

5

6 7

8

9

10

11

12

13

14

15 16

17

27

28

34

35

36

37

38

39

40

41

42

43 44

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.

**Example 12.2** Process 0 consists of two threads. The first thread executes a blocking send call MPI\_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI\_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

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 process threads have completed their MPI calls, and have no pending communications or I/O operations.

Rationale. This constraint simplifies implementation. (End of rationale.)

Multiple threads completing the same request. A program in which 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 can only be completed once. Any combination of wait or test that violates this rule is erroneous.

*Rationale.* 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

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

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.*)

<sup>7</sup> Probe A receive call that uses source and tag values returned by a preceding call to <sup>8</sup> MPI\_PROBE or MPI\_IPROBE will receive the message matched by the probe call only <sup>9</sup> if there was no other matching receive after the probe and before that receive. In a multi-<sup>10</sup> threaded environment, it is up to the user to enforce this condition using suitable mutual <sup>11</sup> exclusion logic. This can be enforced by making sure that each communicator is used by <sup>12</sup> only one thread on each process. Alternatively, MPI\_MPROBE or MPI\_IMPROBE can be <sup>13</sup> used.

14

20

21

22 23

 $^{24}$ 

25

30

31

32

33 34

38

39

40

41

42 43

48

1

 $\mathbf{2}$ 

3

4

5

6

<sup>15</sup> Collective calls Matching of collective calls on a communicator, window, or file handle is <sup>16</sup> done according to the order in which the calls are issued at each process. If concurrent <sup>17</sup> threads issue such calls on the same communicator, window or file handle, it is up to the <sup>18</sup> 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 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 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 communi cator, but not the communicator itself. Accesses to communicators, window handles
 and file handles cannot affect one another. (*End of rationale.*)

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 at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "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 to initialize the MPI thread environment, instead of MPI\_INIT.

#### MPI\_INIT\_THREAD(required, provided)

advice to users.)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)
int MPI_I	nit_thread(int *argc, cha	r ***argv, int required, int *provided)
INTEG INTEG	thread(required, provided ER, INTENT(IN) :: requir ER, INTENT(OUT) :: provi ER, OPTIONAL, INTENT(OUT)	ed ded
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR) INTEGER REQUIRED, PROVIDED, IERROR		
	,	of argc and argv is optional, as with MPI_INIT as II pointers may be passed in their place. ( <i>End of</i>

This call initializes MPI in the same way that a call to MPI\_INIT would. In addition, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

MPI\_THREAD\_SINGLE Only one thread will execute.

MPI\_THREAD\_FUNNELED The process may be multi-threaded, but the application must ensure that only the main thread makes MPI calls (for the definition of main thread, see MPI\_IS\_THREAD\_MAIN on page 487).

 $\mathbf{2}$ 

 $\overline{7}$ 

- 1 **MPI\_THREAD\_SERIALIZED** The process may be multi-threaded, and multiple threads may  $\mathbf{2}$ make MPI calls, but only one at a time: MPI calls are not made concurrently from 3 two distinct threads (all MPI calls are "serialized"). 4 **MPI\_THREAD\_MULTIPLE** Multiple threads may call MPI, with no restrictions. 56 These values are monotonic; i.e., MPI\_THREAD\_SINGLE < MPI\_THREAD\_FUNNELED <  $\overline{7}$ MPI\_THREAD\_SERIALIZED < MPI\_THREAD\_MULTIPLE. 8 Different processes in MPI\_COMM\_WORLD may require different levels of thread sup-9 port. 10 The call returns in provided information about the actual level of thread support that 11will be provided by MPI. It can be one of the four values listed above. 12The level(s) of thread support that can be provided by MPI\_INIT\_THREAD will depend 13on the implementation, and may depend on information provided by the user before the 14program started to execute (e.g., with arguments to mpiexec). If possible, the call will 15return provided = required. Failing this, the call will return the least supported level such 16that provided > required (thus providing a stronger level of support than required by the 17user). Finally, if the user requirement cannot be satisfied, then the call will return in 18 provided the highest supported level. 19A thread compliant MPI implementation will be able to return provided 20= MPI\_THREAD\_MULTIPLE. Such an implementation may always return provided 21= MPI\_THREAD\_MULTIPLE, irrespective of the value of required. 22 An MPI library that is not thread compliant must always return 23provided=MPI\_THREAD\_SINGLE, even if MPI\_INIT\_THREAD is called on a multithreaded  $^{24}$ process. The library should also return correct values for the MPI calls that can be executed 25before initialization, even if multiple threads have been spawned. 2627Such code is erroneous, but if the MPI initialization is performed by a Rationale. 28library, the error cannot be detected until MPI\_INIT\_THREAD is called. The require-29ments in the previous paragraph ensure that the error can be properly detected. (End 30 of rationale.)  $^{31}$ 32 A call to MPI\_INIT has the same effect as a call to MPI\_INIT\_THREAD with a required 33 = MPI\_THREAD\_SINGLE. 34Vendors may provide (implementation dependent) means to specify the level(s) of 35 thread support available when the MPI program is started, e.g., with arguments to mpiexec. 36 This will affect the outcome of calls to MPI\_INIT and MPI\_INIT\_THREAD. Suppose, for 37 example, that an MPI program has been started so that only MPI\_THREAD\_MULTIPLE is 38 available. Then MPI\_INIT\_THREAD will return provided = MPI\_THREAD\_MULTIPLE, irre-39 spective of the value of required; a call to MPI\_INIT will also initialize the MPI thread support 40level to MPI\_THREAD\_MULTIPLE. Suppose, instead, that an MPI program has been started  $^{41}$ so that all four levels of thread support are available. Then, a call to MPI\_INIT\_THREAD 42will return provided = required; alternatively, a call to MPI\_INIT will initialize the MPI 43thread support level to MPI\_THREAD\_SINGLE. 4445Rationale. Various optimizations are possible when MPI code is executed single-
- threaded, 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 ean 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 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 or 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.

Note that required need not be the same value on all processes of MPI\_COMM\_WORLD. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.

		25
MPI_QUERY_THREAD(provided)		26
OUT provided	provided level of thread support (integer)	27
		28
<pre>int MPI_Query_thread(int *provided</pre>	1)	29
MPI_Query_thread(provided, ierror)	BIND(C)	30
INTEGER, INTENT(OUT) :: provi		31
INTEGER, OPTIONAL, INTENT(OUT)		32
		33
MPI_QUERY_THREAD(PROVIDED, IERROR)		34
INTEGER PROVIDED, IERROR		35
The call returns in <b>provided</b> the curr	ent level of thread support, which will be the value	36
returned in provided by MPI_INIT_THREAD, if MPI was initialized by a call to		37
MPI_INIT_THREAD().		38
		39
		40
MPI_IS_THREAD_MAIN(flag)		41
OUT flag	true if calling thread is main thread, false otherwise	42
	(logical)	43
	(1051001)	44
int MDI Is thread main (int uflag)		45
<pre>int MPI_Is_thread_main(int *flag)</pre>		46
<pre>MPI_Is_thread_main(flag, ierror) E</pre>	BIND(C)	47
LOGICAL, INTENT(OUT) :: flag		48

 $\mathbf{2}$ 

 $\overline{7}$ 

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2	MDT TO THDEAD MAIN/ELAC TEDDOD)
3	MPI_IS_THREAD_MAIN(FLAG, IERROR) LOGICAL FLAG
4	INTEGER IERROR
5	INIEGER IERRUR
6	This function can be called by a thread to determine if it is the main thread (the thread
7	that called MPI_INIT or MPI_INIT_THREAD).
8	All routines listed in this section must be supported by all MPI implementations.
9	
10	Rationale. MPI libraries are required to provide these calls even if they do not
11	support threads, so that portable code that contains invocations to these functions
12	can link correctly. MPI_INIT continues to be supported so as to provide compatibility
13	with current MPI codes. (End of rationale.)
14	
15	Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI
16	call other than MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should
17	be executed by these threads, until MPI_INIT_THREAD is invoked by one thread
18	(which, thereby, becomes the main thread). In particular, it is possible to enter the
19	MPI execution with a multi-threaded process.
20	The level of thread support provided is a global property of the MPI process that can
21	be specified only once, when MPI is initialized on that process (or before). Portable
22	third party libraries have to be written so as to accommodate any provided level of
23	thread support. Otherwise, their usage will be restricted to specific level(s) of thread
24	support. If such a library can run only with specific level(s) of thread support, e.g.,
25 26	only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check
20 27	whether the user initialized MPI to the correct level of thread support and, if not,
28	raise an exception. (End of advice to users.)
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

## 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, 15, 48, 52, 58], 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:
t to the second
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 13.2 is
 the position of the eighth etype in the file after the displacement. An "explicit offset"
 is an offset that is used as an argument in explicit data access routines.

- 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	new file handle (handle)	26
001	111	new me nancie (nancie)	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 44provide the same value for amode, and all processes must provide filenames that reference 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 are raised using the default file error handler (see Section 13.9 on page 550). A process can

491

1  $\mathbf{2}$ 

3

4

5

6

7

8

9 10

11

12

13 14

1516

17 18 19

20

28

29

30 31

32

33

34

35

36

37

38

39

40

41 42

43

45

46

47

1 open a file independently of other processes by using the MPI\_COMM\_SELF communicator.  $\mathbf{2}$ The file handle returned, fh, can be subsequently used to access the file until the file is 3 closed using MPI\_FILE\_CLOSE. Before calling MPI\_FINALIZE, the user is required to close 4 (via MPI\_FILE\_CLOSE) all files that were opened with MPI\_FILE\_OPEN. Note that the  $\mathbf{5}$ communicator comm is unaffected by MPI\_FILE\_OPEN and continues to be usable in all 6 MPI routines (e.g., MPI\_SEND). Furthermore, the use of comm will not interfere with I/O 7behavior. 8 The format for specifying the file name in the filename argument is implementation 9 dependent and must be documented by the implementation. 10 An implementation may require that filename include a Advice to implementors. 11 string or strings specifying additional information about the file. Examples include 12the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of 13 machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End 14of advice to implementors.) 1516On some implementations of MPI, the file namespace may not be Advice to users. 17 identical from all processes of all applications. For example, "/tmp/foo" may denote 18 different files on different processes, or a single file may have many names, dependent 19on process location. The user is responsible for ensuring that a single file is referenced 20by the filename argument, as it may be impossible for an implementation to detect 21this type of namespace error. (End of advice to users.) 2223Initially, all processes view the file as a linear byte stream, and each process views data  $^{24}$ in its own native representation (no data representation conversion is performed). (POSIX 25files are linear byte streams in the native representation.) The file view can be changed via 26the MPI\_FILE\_SET\_VIEW routine. 27The following access modes are supported (specified in amode, a bit vector OR of the 28following integer constants): 29• MPI\_MODE\_RDONLY — read only, 30 31• MPI\_MODE\_RDWR — reading and writing, 32 • MPI\_MODE\_WRONLY — write only, 33 34 • MPI\_MODE\_CREATE — create the file if it does not exist, 35 36 • MPI\_MODE\_EXCL — error if creating file that already exists, 37 • MPI\_MODE\_DELETE\_ON\_CLOSE — delete file on close, 38 39 • MPI\_MODE\_UNIQUE\_OPEN — file will not be concurrently opened elsewhere, 40 • MPI\_MODE\_SEQUENTIAL — file will only be accessed sequentially, 41 42• MPI\_MODE\_APPEND — set initial position of all file pointers to end of file. 43 44Advice to users. C users can use bit vector OR () to combine these constants; Fortran 4590 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) 46bit vector IOR on systems that support it. Alternatively, Fortran users can portably 47 use integer addition to OR the constants (each constant should appear at most once 48 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 [39]. 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 on page 498). 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 13.8.1 on page 540). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI\_FILE\_SET\_ATOMICITY.

13.2.2 Closing a File

MPI_FILE_CLOSE(fh)		
INOUT	fh	

file handle (handle)

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

```
1
     int MPI_File_close(MPI_File *fh)
\mathbf{2}
     MPI_File_close(fh, ierror) BIND(C)
3
          TYPE(MPI_File), INTENT(INOUT) :: fh
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     MPI_FILE_CLOSE(FH, IERROR)
\overline{7}
          INTEGER FH, IERROR
8
          MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
9
      MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
10
      opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
11
      MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
12
13
           Advice to users. If the file is deleted on close, and there are other processes currently
14
           accessing the file, the status of the file and the behavior of future accesses by these
15
           processes are implementation dependent. (End of advice to users.)
16
17
          The user is responsible for ensuring that all outstanding nonblocking requests and
18
      split collective operations associated with fh made by a process have completed before that
19
      process calls MPI_FILE_CLOSE.
20
          The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
21
     MPI_FILE_NULL.
22
23
      13.2.3 Deleting a File
^{24}
25
26
      MPI_FILE_DELETE(filename, info)
27
       IN
                                              name of file to delete (string)
                  filename
28
29
       IN
                 info
                                              info object (handle)
30
^{31}
      int MPI_File_delete(const char *filename, MPI_Info info)
32
33
     MPI_File_delete(filename, info, ierror) BIND(C)
34
          CHARACTER(LEN=*), INTENT(IN) :: filename
35
          TYPE(MPI_Info), INTENT(IN) :: info
36
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
37
     MPI_FILE_DELETE(FILENAME, INFO, IERROR)
38
          CHARACTER*(*) FILENAME
39
          INTEGER INFO, IERROR
40
41
          MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does
42
      not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.
43
          The info argument can be used to provide information regarding file system specifics
     (see Section 13.2.8 on page 498). The constant MPI_INFO_NULL refers to the null info, and
44
45
      can be used when no info needs to be specified.
46
          If a process currently has the file open, the behavior of any access to the file (as well
47
      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

494

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 13.9 on page 550).

Resizing a File 13.2.4

		6
MPI_FILE_SET_SIZE(fh, size)		7
		8
INOUT fh	file handle (handle)	9
IN size	size to truncate or expand file (integer)	10
	- ( ~ ~ )	11
int MPI_File_set_size(MPI_File fh	MPI Offset size)	12
110 11 1_1 110_000_0120(11 1_1 110 11	,	13
MPI_File_set_size(fh, size, ierro	r) BIND(C)	14
<pre>TYPE(MPI_File), INTENT(IN) ::</pre>	fh	15
<pre>INTEGER(KIND=MPI_OFFSET_KIND)</pre>	, INTENT(IN) :: size	16
INTEGER, OPTIONAL, INTENT(OUT	) :: ierror	17
MPI_FILE_SET_SIZE(FH, SIZE, IERRO	ומ	18
INTEGER FH, IERROR	n)	19
INTEGER (KIND=MPI OFFSET KIND)	ST7F	20
INTEGEN(NIND=HFI_OFFBEI_KIND)		21

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.

It is possible for the file pointers to point beyond the end of file Advice to users. after a MPI\_FILE\_SET\_SIZE operation truncates a file. This is valid, and equivalent to seeking beyond the current end of file. (End of advice to users.)

All nonblocking requests and split collective operations on fh must be completed before calling MPI\_FILE\_SET\_SIZE. Otherwise, calling MPI\_FILE\_SET\_SIZE is erroneous. As far as consistency semantics are concerned, MPI\_FILE\_SET\_SIZE is a write operation that conflicts with operations that access bytes at displacements between the old and new file sizes (see Section 13.8.1 on page 540).

1

2

3 4

5

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34 35

36

37

38 39

40

41

42

43

```
1
     13.2.5
              Preallocating Space for a File
\mathbf{2}
3
4
      MPI_FILE_PREALLOCATE(fh, size)
5
       INOUT
                 fh
                                               file handle (handle)
6
7
       IN
                 size
                                              size to preallocate file (integer)
8
9
      int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
10
     MPI_File_preallocate(fh, size, ierror) BIND(C)
11
          TYPE(MPI_File), INTENT(IN) ::
                                               fh
12
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                                size
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
14
15
     MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
16
          INTEGER FH, IERROR
17
          INTEGER(KIND=MPI_OFFSET_KIND) SIZE
18
          MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes
19
      of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the
20
      group must pass identical values for size. Regions of the file that have previously been
21
      written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE
22
      has the same effect as writing undefined data. If size is larger than the current file size, the
23
      file size increases to size. If size is less than or equal to the current file size, the file size is
24
      unchanged.
25
          The treatment of file pointers, pending nonblocking accesses, and file consistency is the
26
      same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when
27
      the file was opened, it is erroneous to call this routine.
28
29
           Advice to users. In some implementations, file preallocation may be expensive. (End
30
           of advice to users.)
^{31}
32
     13.2.6 Querying the Size of a File
33
34
35
      MPI_FILE_GET_SIZE(fh, size)
36
37
       IN
                  fh
                                               file handle (handle)
38
       OUT
                                               size of the file in bytes (integer)
                 size
39
40
      int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
^{41}
42
     MPI_File_get_size(fh, size, ierror) BIND(C)
43
          TYPE(MPI_File), INTENT(IN) :: fh
44
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::
                                                                  size
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
          INTEGER FH, IERROR
48
```

MPI the file h data acc		<b>ze</b> , the current size in bytes of the file associated with semantics are concerned, MPI_FILE_GET_SIZE is a	1 2 3 4 5 6 7 8 9
MPI_FIL	E_GET_GROUP(fh, group)		10
IN	fh	file handle (handle)	11
OUT	group	group which opened the file (handle)	12 13 14
int MPI	_File_get_group(MPI_File d	fh, MPI_Group *group)	14 15
TYP: TYP:	e_get_group(fh, group, ien E(MPI_File), INTENT(IN) : E(MPI_Group), INTENT(OUT) EGER, OPTIONAL, INTENT(OUT)	: fh :: group	16 17 18 19 20
	MPI_FILE_GET_GROUP(FH, GROUP, IERROR) INTEGER FH, GROUP, IERROR		20 21 22
	file associated with fh. The gr	duplicate of the group of the communicator used to roup is returned in <b>group</b> . The user is responsible for	23 24 25 26 27
MPI_FIL	E_GET_AMODE(fh, amode)		28
IN	fh	file handle (handle)	29 30
OUT	amode	file access mode used to open the file (integer)	30 31 32
int MPI	_File_get_amode(MPI_File d	fh, int *amode)	33
TYP: INT	<pre>MPI_File_get_amode(fh, amode, ierror) BIND(C)     TYPE(MPI_File), INTENT(IN) :: fh     INTEGER, INTENT(OUT) :: amode     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>		34 35 36 37 38
	E_GET_AMODE(FH, AMODE, IE EGER FH, AMODE, IERROR	RROR)	39 40
MPI fh.	_FILE_GET_AMODE returns, i	in amode, the access mode of the file associated with	41 42 43
Example the follow		ng an <b>amode</b> bit vector will require a routine such as	44 45 46

```
1
            SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
\mathbf{2}
     ļ
3
     !
         TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
4
     !
         IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
5
     Ţ
6
            INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
7
            BIT_FOUND = 0
8
            CP_AMODE = AMODE
9
      100 CONTINUE
10
            LBIT = 0
11
            HIFOUND = 0
12
            DO 20 L = MAX_BIT, 0, -1
13
               MATCHER = 2**L
14
               IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
15
                  HIFOUND = 1
16
                  LBIT = MATCHER
17
                  CP_AMODE = CP_AMODE - MATCHER
18
               END IF
19
       20
           CONTINUE
20
            IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
21
            IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
22
                CP_AMODE .GT. 0) GO TO 100
23
            END
24
         This routine could be called successively to decode amode, one bit at a time. For
25
     example, the following code fragment would check for MPI_MODE_RDONLY.
26
27
            CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
28
            IF (BIT_FOUND .EQ. 1) THEN
29
               PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
30
            ELSE
31
               PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
32
            END IF
33
34
     13.2.8 File Info
35
36
     Hints specified via info (see Chapter 9 on page 365) allow a user to provide information
37
38
```

<sup>37</sup> such as file access patterns and file system specifics to direct optimization. Providing
<sup>38</sup> hints may enable an implementation to deliver increased I/O performance or minimize the
<sup>39</sup> use of system resources. However, hints do not change the semantics of any of the I/O
<sup>40</sup> interfaces. In other words, an implementation is free to ignore all hints. Hints are specified
<sup>41</sup> on a per file basis, in MPI\_FILE\_OPEN, MPI\_FILE\_DELETE, MPI\_FILE\_SET\_VIEW, and
<sup>42</sup> MPI\_FILE\_SET\_INFO, via the opaque info object. When an info object that specifies a
<sup>43</sup> subset of valid hints is passed to MPI\_FILE\_SET\_VIEW or MPI\_FILE\_SET\_INFO, there will
<sup>44</sup> 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)
           fh
 INOUT
                                     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
```

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.

Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. (*End of advice to users.*)

MPI\_FILE\_GET\_INFO(fh, info\_used)

IN	fh	file handle (handle)	32
OUT	info_used	new info object (handle)	33
001			34
int MDT T	Cile met info(MDI File fb	MDI Info winfo used)	35
IIIC MPI_F	File_get_info(MPI_File fh,	MP1_1110 *11110_used)	36
MPI_File_	_get_info(fh, info_used, i	error) BIND(C)	37
TYPE	(MPI_File), INTENT(IN) ::	fh	38
TYPE	(MPI_Info), INTENT(OUT) ::	info_used	39
INTEC	ER, OPTIONAL, INTENT(OUT)	:: ierror	40
WET ETTE			41
	GET_INFO(FH, INFO_USED, ]	ERRUR)	42
INTEC	ER FH, INFO_USED, IERROR		43
MPL I	- 	v info object containing the hints of the file associ-	44

MPI\_FILE\_GET\_INFO returns a new info object containing the hints of the file associated with fh. The current setting of all hints actually used by the system related to this open file is returned in info\_used. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pairs. The user is responsible for freeing info\_used via MPI\_INFO\_FREE. 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

1

 $\mathbf{2}$ 

3

4

5

6 7

 $^{31}$ 

32

33

34 35

36

37

38

39

Advice to users. The info object returned in info\_used will contain all hints currently active for this file. This set of hints may be greater or smaller than the set of hints passed in to MPI\_FILE\_OPEN, MPI\_FILE\_SET\_VIEW, or MPI\_FILE\_SET\_INFO, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (*End of advice to users.*)

#### Reserved File Hints

Some potentially useful hints (info key values) are outlined below. 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. (For more details on "info," see Chapter 9 on page 365.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., file\_perm is only useful during file creation).

access\_style (comma separated list of strings): This hint specifies the manner in which
 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.

<sup>24</sup> collective\_buffering (boolean) [SAME]: This hint specifies whether the application may
 <sup>25</sup> benefit from collective buffering. Collective buffering is an optimization performed
 <sup>26</sup> on collective accesses. Accesses to the file are performed on behalf of all processes in
 <sup>27</sup> the group by a number of target nodes. These target nodes coalesce small requests
 <sup>28</sup> into large disk accesses. Valid values for this key are true and false. Collective buffering
 <sup>29</sup> parameters are further directed via additional hints: cb\_block\_size, cb\_buffer\_size, and
 <sup>30</sup> 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).
- <sup>47</sup> chunked\_item (comma separated list of integers) [SAME]: This hint specifies the size
   <sup>48</sup> 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.
- num\_io\_nodes (integer) [SAME]: This hint specifies the number of I/O devices in the system. This hint is most relevant when the file is created.
- striping\_factor (integer) [SAME]: This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.
- striping\_unit (integer) [SAME]: This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

#### 13.3 File Views

			36
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)		etype, datarep, info)	37
INOUT	fh	file handle (handle)	38
IN	disp	displacement (integer)	39
	•		40
IN	etype	elementary datatype (handle)	41
IN	filetype	filetype (handle)	42
IN	datarep	data representation (string)	43
	·	data representation (string)	44
IN	info	info object (handle)	45
			46

 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

47

1	MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)
2	TYPE(MPI_File), INTENT(IN) :: fh
3	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
4	TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
5	CHARACTER(LEN=*), INTENT(IN) :: datarep
6	TYPE(MPI_Info), INTENT(IN) :: info
7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8	inidelit, official, initial (001) icitor
9	MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
10	INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
11	CHARACTER*(*) DATAREP
12	INTEGER(KIND=MPI_OFFSET_KIND) DISP
13	
14	The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file.
15	The start of the view is set to disp; the type of data is set to etype; the distribution of data
	to processes is set to filetype; and the representation of data in the file is set to datarep.
16	In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file
17	pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents
18	of etype in the file data representation must be identical on all processes in the group; values
19	for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be
20	committed.
21	The etype always specifies the data layout in the file. If etype is a portable datatype
22	(see Section 2.4 on page 11), the extent of etype is computed by scaling any displacements
23	in the datatype to match the file data representation. If etype is not a portable datatype,
24	no scaling is done when computing the extent of etype. The user must be careful when
25	using nonportable etypes in heterogeneous environments; see Section 13.7.1 on page 532 for
26	further details.
27	If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special
28	displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displace-
29	ment to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is
30	invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.
31	
32	Rationale. For some sequential files, such as those corresponding to magnetic tapes
33	or streaming network connections, the <i>displacement</i> may not be meaningful.
34	MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files.
35	(End of rationale.)
36	
37	Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will
38	immediately follow MPI_FILE_OPEN in numerous instances. A high-quality imple-
39	mentation will ensure that this behavior is efficient. (End of advice to implementors.)
40	
41	The disp displacement argument specifies the position (absolute offset in bytes from
42	the beginning of the file) where the view begins.
43	
44	Advice to users. disp can be used to skip headers or when the file includes a sequence
45	of data segments that are to be accessed in different patterns (see Figure $13.3$ ). Sep-
46	arate views, each using a different displacement and filetype, can be used to access
47	each segment.
48	(End of advice to users.)

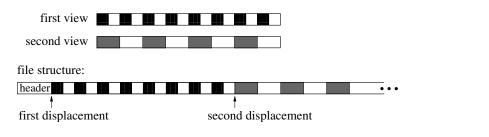


Figure 13.3: Displacements

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.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (see Section 13.6 on page 530). (End of advice to users.)

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. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain 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 processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype, and filetype arguments can be changed via future calls to MPI\_FILE\_SET\_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype.

The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8 on page 498). The constant MPI\_INFO\_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 13.6 on page 530) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI\_FILE\_SET\_VIEW — otherwise, the call to MPI\_FILE\_SET\_VIEW is erroneous.

 $^{24}$ 

 $^{31}$ 

```
1
     MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)
2
       IN
                fh
                                            file handle (handle)
3
       OUT
                disp
                                            displacement (integer)
4
5
       OUT
                                            elementary datatype (handle)
                etype
6
       OUT
                filetype
                                            filetype (handle)
7
       OUT
                datarep
                                            data representation (string)
8
9
     int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
10
                    MPI_Datatype *filetype, char *datarep)
11
12
     MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) BIND(C)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::
                                                             disp
15
         TYPE(MPI_Datatype), INTENT(OUT) ::
                                                 etype, filetype
16
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
17
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
18
19
     MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
          INTEGER FH, ETYPE, FILETYPE, IERROR
20
21
         CHARACTER*(*) DATAREP
22
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
23
```

<sup>24</sup> MPI\_FILE\_GET\_VIEW returns the process's view of the data in the file. The current <sup>25</sup> value of the displacement is returned in disp. The etype and filetype are new datatypes with <sup>26</sup> 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.

33 34 35

36

37

27

28

29

30

 $^{31}$ 

32

### 13.4 Data Access

#### 13.4.1 Data Access Routines

<sup>38</sup> Data is moved between files and processes by issuing read and write calls. There are <sup>40</sup> three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), <sup>41</sup> synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective <sup>42</sup> *vs.* collective). The following combinations of these data access routines, including two <sup>43</sup> types of file pointers (individual and shared) are provided in Table 13.1.

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI\_FILE\_READ and MPI\_FILE\_WRITE.

Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation

positioning	synchronism		coordination
		noncollective	collective
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	nonblocking &	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN
	split collective		MPI_FILE_READ_AT_ALL_END
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL
	nonblocking &	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN
	split collective		MPI_FILE_READ_ALL_END
		MPI_FILE_IWRITE	MPI_FILE_WRITE_ALL_BEGIN
			MPI_FILE_WRITE_ALL_END
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED
	nonblocking &	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN
	split collective		MPI_FILE_READ_ORDERED_END
		MPI_FILE_IWRITE_SHARED	MPI_FILE_WRITE_ORDERED_BEGIN
			MPI_FILE_WRITE_ORDERED_END

Table 13.1: Data access routines

completes. For writes, however, the MPI\_FILE\_SYNC routine provides the only guarantee that data has been transferred to the storage device.

#### Positioning

MPI provides three types of positioning for data access routines: explicit offsets, individual file pointers, and shared file pointers. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain \_AT in their name (e.g., MPI\_FILE\_WRITE\_AT). Explicit offset operations perform data access at the file position given directly as an argument — no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. Operations with explicit offsets are described in Section 13.5.1 on page 507.

The names of the individual file pointer routines contain no positional qualifier (e.g., MPI\_FILE\_WRITE). Operations with individual file pointers are described in Section 13.5.2 on page 511. The data access routines that use shared file pointers contain \_SHARED or \_ORDERED in their name (e.g., MPI\_FILE\_WRITE\_SHARED). Operations with shared file pointers are described in

#### 13.5

#### 13.5.3.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

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

 $^{24}$ 

 $^{31}$ 

offset before the call. The file position, new\_file\_offset, is in terms of a count of etypes  $\mathbf{2}$ relative to the current view.

3 4

7

1

Synchronism

 $\mathbf{5}$ MPI supports blocking and nonblocking I/O routines. 6

A blocking I/O call will not return until the I/O request is completed.

A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. 8 Given suitable hardware, this allows the transfer of data out of and into the user's buffer 9 to proceed concurrently with computation. A separate request complete call (MPI\_WAIT, 10 MPI\_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm 11 that the data has been read or written and that it is safe for the user to reuse the buffer. 12The nonblocking versions of the routines are named MPI\_FILE\_IXXX, where the I stands 13 for immediate. 14

It is erroneous to access the local buffer of a nonblocking data access operation, or to 15use that buffer as the source or target of other communications, between the initiation and 16completion of the operation. 17

The split collective routines support a restricted form of "nonblocking" operations for 18 collective data access (see Section 13.5.4 on page 523). 19

#### 20

#### Coordination 21

22Every noncollective data access routine MPI\_FILE\_XXX has a collective counterpart. For 23most routines, this counterpart is MPI\_FILE\_XXX\_ALL or a pair of MPI\_FILE\_XXX\_BEGIN  $^{24}$ and MPI\_FILE\_XXX\_END. The counterparts to the MPI\_FILE\_XXX\_SHARED routines are 25MPI\_FILE\_XXX\_ORDERED. 26

The completion of a noncollective call only depends on the activity of the calling pro-27cess. However, the completion of a collective call (which must be called by all members of 28the process group) may depend on the activity of the other processes participating in the 29collective call. See Section 13.8.4 on page 544 for rules on semantics of collective calls. 30

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

32 33

34

 $^{31}$ 

#### Data Access Conventions

35 Data is moved between files and processes by calling read and write routines. Read routines 36 move data from a file into memory. Write routines move data from memory into a file. The 37 file is designated by a file handle, fh. The location of the file data is specified by an offset 38into the current view. The data in memory is specified by a triple: buf, count, and datatype. 39 Upon completion, the amount of data accessed by the calling process is returned in a status.

40An 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  $^{41}$ 42(negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI. 43

A data access routine attempts to transfer (read or write) count data items of type 4445datatype between the user's buffer buf and the file. The datatype passed to the routine 46must be a committed datatype. The layout of data in memory corresponding to buf, count, 47datatype is interpreted the same way as in MPI communication functions; see Section 3.2.248on page 25 and Section 4.1.11 on page 111. The data is accessed from those parts of the file 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 Sections 17.1.10–17.1.20. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS (or MPI\_GET\_ELEMENTS\_X), respectively. The interpretation of the MPI\_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI\_ERR\_IN\_STATUS. The user can pass (in C and Fortran) MPI\_STATUS\_IGNORE in the status argument if the return value of this argument is not needed. The status can be passed to MPI\_TEST\_CANCELLED to determine if the operation was cancelled. All other fields of status are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

#### 13.5.1 Data Access with Explicit Offsets

If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section.

MPI\_FILE\_READ\_AT(fh, offset, buf, count, datatype, status)

	a		35		
IN	fh	file handle (handle)	36		
IN	offset	file offset (integer)	37		
OUT	buf	initial address of buffer (choice)	38		
IN	count	number of elements in buffer (integer)	39		
	count	number of elements in builer (meeser)	40		
IN	datatype	datatype of each buffer element (handle)	41		
OUT	status	status object (Status)	42		
			43		
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)					

MPI\_File\_read\_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
 TYPE(MPI\_File), INTENT(IN) :: fh

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

508
-----

```
1
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
\mathbf{2}
         TYPE(*), DIMENSION(..) :: 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_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
8
         <type> BUF(*)
9
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
10
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
11
12
         MPI_FILE_READ_AT reads a file beginning at the position specified by offset.
13
14
     MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)
15
16
       IN
                fh
                                            file handle (handle)
17
                offset
       IN
                                            file offset (integer)
18
       OUT
                buf
                                            initial address of buffer (choice)
19
20
       IN
                count
                                            number of elements in buffer (integer)
21
       IN
                datatype
                                            datatype of each buffer element (handle)
22
       OUT
                status
                                            status object (Status)
23
^{24}
25
     int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
26
                    int count, MPI_Datatype datatype, MPI_Status *status)
27
     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
28
                    BIND(C)
29
         TYPE(MPI_File), INTENT(IN) :: fh
30
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
31
         TYPE(*), DIMENSION(..) :: buf
32
         INTEGER, INTENT(IN) :: count
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Status) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
38
          <type> BUF(*)
39
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
40
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
41
         MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT
42
     interface.
43
44
45
46
47
48
```

<pre>NPT_FILE_VNTIE_AT(IN, OFSET, BUF, COUNT, GATATYPE, STATUS, IERROR IN offset file beginning at the position specified by offset.</pre>	MPI FILE WRITE AT(th offset but count datatype status) $1$						
<pre>N offset file offset (integer) N buf initial address of buffer (choice) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	Wi 1_1 1EE_Wi(1 E_AT(III, Oliset, but, count, datatype, status)						
<pre>IN buf initial address of Duffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	INOUT		file handle (handle)	3			
<pre>N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	IN	offset	file offset (integer)	4			
<pre>N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	IN	buf	initial address of buffer (choice)				
INdatatypedatatype of each buffer element (handle)sOUTstatusstatus object (Status)9OUTstatusstatus object (Status)9int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)11MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)14TYPE(MPI_File), INTENT(IN) :: fh15INTEGER, KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset16TYPE(MPI_File), INTENT(IN) :: count17INTEGER, NTENT(IN) :: count18TYPE(MPI_Status) :: status10INTEGER, OPTIONAL, INTENT(OUT) :: ierror11MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)22NTEGER (KIND=MPI_OFFSET_KIND) OFFSET23MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.33INOUTfhfile nandle (handle)INoffsetfile offset (integer)INbufinitial address of buffer (choice)INdatatypedatatype of each buffer element (handle)OUTstatusstatus object (Status)int count, MPI_Datatype datatype, MPI_Status *status)34int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)35MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)36MPI_File_writ	IN	count	number of elements in buffer (integer)				
<pre>int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	IN	datatype	datatype of each buffer element (handle)				
<pre>int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	OUT	status	status object (Status)	9			
<pre>int wPI_FITe_WFITE_AT.WPI_OTESt offset, const void *but,</pre>				10			
<pre>MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C) MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C) MPI_File_WrITE_AT(FH, OFFSET, KIND), INTENT(IN) :: offset TYPE(WPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR iNTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status) NOUT fh file handle (handle) N offset file offset (integer) N buf initial address of buffer (choice) N buf initial address of buffer (choice) N datatype datatype of each buffer element (handle) N offset status object (Status) NUT fh. initial address of buffer (choice) N buf initial address of buffer (choice) N buf initial address of buffer (choice) N datatype datatype datatype, MPI_Status *status) MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</type></pre>	int MPI_F	ile_write_at(MPI_File fh,	MPI_Offset offset, const void *buf,				
<pre>MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C) TYPE(MPI_File), INTENT(IN):: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN):: offset TYPE(*), DIMENSION(), INTENT(IN):: buf INTEGER, INTENT(IN):: count TYPE(MPI_Datatype), INTENT(IN):: datatype TYPE(MPI_Status): status INTEGER, OPTIONAL, INTENT(UUT):: ierror MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <pre></pre></pre>		int count, MPI_Dataty	ype datatype, MPI_Status *status)				
<pre>TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTERGEN, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <pre></pre></pre>	MPI_File_	write_at(fh, offset, buf,	count, datatype, status, ierror) BIND(C)				
<pre>TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror  MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	TYPE(]	MPI_File), INTENT(IN) ::	fh				
<pre>INTEGER, INTENT(IN) :: count</pre>				16			
<pre>TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror  MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type>BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status) NOUT fh file handle (handle) N offset file offset (integer) N buf initial address of buffer (choice) N datatype datatype of each buffer element (handle) OUT status status object (Status) NPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</type></pre>			'(IN) :: buf	17			
<pre>TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset. MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN datatype datatype of each buffer element (handle) OUT status status object (Status) int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</type></pre>		-	· · datatype				
<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror 21 MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>		• -					
<pre>MP1_FILE_wRITE_AT(FH, OFFSET, BOF, COUNT, DATATYPE, STATUS, TERROR)</pre>			:: ierror				
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset. MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) INOUT status tatus object (Status) MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count</type>	MPT FTLE	WRITE AT(FH. OFFSET. BUF.	COUNT, DATATYPE, STATUS, TEBROR)	22			
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_STAEL), TERROR       25         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET       26         MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.       27         MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)       30         INOUT       fh       file handle (handle)       31         IN       offset       file offset (integer)       32         IN       buf       initial address of buffer (choice)       33         IN       count       number of elements in buffer (integer)       34         IN       datatype       datatype of each buffer element (handle)       36         OUT       status       status object (Status)       38         int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)       41         MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)       38         BIND(C)       10       11         TYPE(MPI_File), INTENT(IN) ::       fh       44         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::       offset         TYPE(*), DIMENSION(), INTENT(IN) ::       buf       46         INTEGER, INTENT(IN) ::       count       47				23			
INTEGER(KIND=MPI_OFFSEI_KIND) OFFSEI       26         MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.       27         MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)       30         INOUT       fh       file handle (handle)       31         IN       offset       file offset (integer)       32         IN       buf       initial address of buffer (choice)       33         IN       count       number of elements in buffer (integer)       34         IN       datatype       datatype of each buffer element (handle)       36         OUT       status       status object (Status)       37         int       count, MPI_Datatype datatype, MPI_Status *status)       41         MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)       42         BIND(C)       43       44         TYPE(MPI_File), INTENT(IN) ::       fh       44         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::       offset       45         TYPE(*), DIMENSION(), INTENT(IN) ::       buf       46         INTEGER, INTENT(IN) ::       count       47	INTEG	ER FH, COUNT, DATATYPE, S	TATUS(MPI_STATUS_SIZE), IERROR				
<pre>MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset. 27 NPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status) NOUT fh file handle (handle) 31 NOUT fh file offset (integer) 32 N buf initial address of buffer (choice) 34 N count number of elements in buffer (integer) 35 N datatype datatype of each buffer element (handle) 36 OUT status status object (Status) 37 OUT status status object (Status) 38 int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, 40 int count, MPI_Datatype datatype, MPI_Status *status) 41 MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) BIND(C) 43 TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 45 TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count 47</pre>	INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET				
MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)       29         INOUT       fh       file handle (handle)       31         IN       offset       file offset (integer)       32         IN       buf       initial address of buffer (choice)       33         IN       count       number of elements in buffer (integer)       35         IN       datatype       datatype of each buffer element (handle)       36         OUT       status       status object (Status)       38         int       MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)       39         MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)       41         MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)       42         BIND(C)       TYPE(MPI_File), INTENT(IN) :: fh       44         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset       45         TYPE(*), DIMENSION(), INTENT(IN) :: buf       46         INTEGER, INTENT(IN) :: count       47	MPI_F	ILE_WRITE_AT writes a file h	beginning at the position specified by offset.				
MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)30INOUTfhfile handle (handle)31INoffsetfile offset (integer)32INbufinitial address of buffer (choice)33INcountnumber of elements in buffer (integer)35INdatatypedatatype of each buffer element (handle)36OUTstatusstatus object (Status)37intMPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)39MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)41BIND(C)43TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset44INTEGER, INTENT(IN) :: count47				28			
INOUTfhfile handle (handle)30INoffsetfile offset (integer)32INbufinitial address of buffer (choice)33INcountnumber of elements in buffer (integer)35INdatatypedatatype of each buffer element (handle)36OUTstatusstatus object (Status)37intMPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)39MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)42BIND(C)43TYPE(MPI_File), INTENT(IN) ::ffsetTYPE(*), DIMENSION(), INTENT(IN) ::offsetINTEGER, INTENT(IN) ::countMTEGER, INTENT(IN) ::count47	MPI_FILE_	WRITE_AT_ALL(fh, offset, bu	f, count, datatype, status)				
<pre>IN offset file offset (integer) 32 IN buf initial address of buffer (choice) 33 IN count number of elements in buffer (integer) 35 IN datatype datatype of each buffer element (handle) 36 OUT status status object (Status) 37 int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 41 MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) 42 BIND(C) 43 TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 45 TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count 47</pre>		· ·	··· ,				
<pre>N buf initial address of buffer (choice) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) N PI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>							
INcountnumber of elements in buffer (integer)34INdatatypedatatype of each buffer element (handle)35OUTstatusstatus object (Status)37OUTstatusstatus object (Status)38intMPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)40MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) BIND(C)42TYPE(MPI_File), INTENT(IN) ::fhINTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::offsetTYPE(*), DIMENSION(), INTENT(IN) ::buffINTEGER, INTENT(IN) ::count47				33			
<pre>IN datatype datatype of each buffer element (handle) OUT status status object (Status)  int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>				34			
OUTstatusstatus object (Status)3738int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)39MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) BIND(C)41MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)42MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)42MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)42BIND(C)43TYPE(MPI_File), INTENT(IN) :: fh44INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset45TYPE(*), DIMENSION(), INTENT(IN) :: buf46INTEGER, INTENT(IN) :: count47							
OUTstatusstatus object (Status)38int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)39MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)41MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)42BIND(C)43TYPE(MPI_File), INTENT(IN) :: fh44INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset45TYPE(*), DIMENSION(), INTENT(IN) :: buf46INTEGER, INTENT(IN) :: count47	IN	datatype	datatype of each buffer element (handle)				
<pre>int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	OUT	status	status object (Status)				
<pre>int count, MPI_Datatype datatype, MPI_Status *status)  MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) BIND(C)  TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count 40 41 40 41 40 41 41 41 41 41 41 41 41 41 41 41 41 41</pre>				39			
<pre>MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count 47</pre>	int MPI_F			40			
MF1_FITe_wfite_at_all(In, offset, but, count, datatype, status, ferror)BIND(C)TYPE(MPI_File), INTENT(IN) :: fhINTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offsetTYPE(*), DIMENSION(), INTENT(IN) :: bufINTEGER, INTENT(IN) :: count		int count, MPI_Dataty	/pe datatype, MP1_Status *status/				
TYPE(MPI_File), INTENT(IN) :: fh44INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset45TYPE(*), DIMENSION(), INTENT(IN) :: buf46INTEGER, INTENT(IN) :: count47	MPI_File_		buf, count, datatype, status, ierror)				
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset45TYPE(*), DIMENSION(), INTENT(IN) :: buf46INTEGER, INTENT(IN) :: count47	TVDE (1		fb				
TYPE(*), DIMENSION(), INTENT(IN) :: buf46INTEGER, INTENT(IN) :: count47							
INTEGER, INTENT(IN) COURC							
TYPE(MPI_Datatype), INTENT(IN) :: datatype   48		•					
	TYPE(]	48					

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
4
          <type> BUF(*)
5
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
6
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
7
8
         MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
9
     MPI_FILE_WRITE_AT interface.
10
11
     MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
12
13
       IN
                 fh
                                            file handle (handle)
14
       IN
                offset
                                            file offset (integer)
15
                                            initial address of buffer (choice)
       OUT
                buf
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
22
     int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
23
                    MPI_Datatype datatype, MPI_Request *request)
^{24}
     MPI_File_iread_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(..), 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_IREAD_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_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
39
40
41
42
43
44
45
46
47
48
```

MPI_FILE_	_IWRITE_AT(fh, offset, buf, co	unt, datatype, request)	1
INOUT	fh	file handle (handle)	2
IN	offset	file offset (integer)	3
			4
IN	buf	initial address of buffer (choice)	5 6
IN	count	number of elements in buffer (integer)	7
IN	datatype	datatype of each buffer element (handle)	8
OUT	request	request object (handle)	9
		request esject (narate)	10
int MPT F	'ile iwrite at(MPI File f	n, MPI_Offset offset, const void *buf,	11
		ype datatype, MPI_Request *request)	12
			13
MPI_File_		, count, datatype, request, ierror)	14
TYPE	BIND(C)	fh	15
	<pre>MPI_File), INTENT(IN) :: ER(KIND=MPI_OFFSET_KIND)</pre>		16
	-	C(IN), ASYNCHRONOUS :: buf	17 18
	ER, INTENT(IN) :: count		19
	MPI_Datatype), INTENT(IN)	:: datatype	20
	MPI_Request), INTENT(OUT)		21
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	22
MDT ETTE	TUDITE AT/EU OFECET DII	F, COUNT, DATATYPE, REQUEST, IERROR)	23
	<pre>&gt; BUF(*)</pre>	, COONI, DAIAIIPE, REQUESI, IERROR)	24
01	ER FH, COUNT, DATATYPE, H	REDUEST TERROR	25
	ER(KIND=MPI_OFFSET_KIND)		26
			27
MPI_F	FILE_IWRITE_AT is a nonblock	king version of the MPI_FILE_WRITE_AT interface.	28
10 - 0 -			29
13.5.2 D	ata Access with Individual Fil	e Pointers	30
			31

MPI maintains one individual file pointer per process per file handle. The current value of this pointer implicitly specifies the offset in the data access routines described in this 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 13.5.1 on page 507, 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.

```
1
     MPI_FILE_READ(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                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
     int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
10
                    MPI_Status *status)
11
12
     MPI_File_read(fh, buf, count, datatype, status, ierror) BIND(C)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         TYPE(*), DIMENSION(..) :: 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
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
20
          <type> BUF(*)
21
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
22
23
         MPI_FILE_READ reads a file using the individual file pointer.
^{24}
25
     Example 13.2 The following Fortran code fragment is an example of reading a file until
26
     the end of file is reached:
27
28
         Read a preexisting input file until all data has been read.
     !
29
         Call routine "process_input" if all requested data is read.
     Т
30
     I.
         The Fortran 90 "exit" statement exits the loop.
31
32
                       bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
            integer
33
            parameter (bufsize=100)
34
            real
                       localbuffer(bufsize)
35
            integer (kind=MPI_OFFSET_KIND) zero
36
37
            zero = 0
38
39
            call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
40
                                  MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
41
            call MPI_FILE_SET_VIEW( myfh, zero, MPI_REAL, MPI_REAL, 'native', &
42
                                  MPI_INFO_NULL, ierr )
43
            totprocessed = 0
44
            do
45
               call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
46
                                     status, ierr )
47
               call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
48
               call process_input( localbuffer, numread )
```

48

1 totprocessed = totprocessed + numread  $\mathbf{2}$ if ( numread < bufsize ) exit 3 enddo 4 write(6,1001) numread, bufsize, totprocessed 56 1001 format( "No more data: read", I3, "and expected", I3, & "Processed total of", I6, "before terminating job." ) 7 8 call MPI\_FILE\_CLOSE( myfh, ierr ) 9 10 11 12MPI\_FILE\_READ\_ALL(fh, buf, count, datatype, status) 13 INOUT fh file handle (handle) 1415OUT buf initial address of buffer (choice) 16IN number of elements in buffer (integer) count 17IN datatype of each buffer element (handle) 18 datatype 19 OUT status status object (Status) 2021int MPI\_File\_read\_all(MPI\_File fh, void \*buf, int count, 22 MPI\_Datatype datatype, MPI\_Status \*status) 2324MPI\_File\_read\_all(fh, buf, count, datatype, status, ierror) BIND(C) TYPE(MPI\_File), INTENT(IN) :: 25fh 26TYPE(\*), DIMENSION(...) :: buf INTEGER, INTENT(IN) :: count 27TYPE(MPI\_Datatype), INTENT(IN) :: datatype 2829 TYPE(MPI\_Status) :: status 30 INTEGER, OPTIONAL, INTENT(OUT) :: 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 MPI\_FILE\_READ\_ALL is a collective version of the blocking MPI\_FILE\_READ interface. 35 36 37 MPI\_FILE\_WRITE(fh, buf, count, datatype, status) 38 39 INOUT fh file handle (handle) 40 IN buf initial address of buffer (choice) 41 IN count number of elements in buffer (integer) 42IN 43 datatype of each buffer element (handle) datatype 44OUT status status object (Status) 4546int MPI\_File\_write(MPI\_File fh, const void \*buf, int count, 47MPI\_Datatype datatype, MPI\_Status \*status)

```
1
     MPI_File_write(fh, buf, count, datatype, status, ierror) BIND(C)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
4
         INTEGER, INTENT(IN) :: count
\mathbf{5}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Status) :: status
7
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
8
     MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
9
          <type> BUF(*)
10
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
11
12
         MPI_FILE_WRITE writes a file using the individual file pointer.
13
14
     MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
15
16
       INOUT
                 fh
                                            file handle (handle)
17
       IN
                 buf
                                            initial address of buffer (choice)
18
       IN
                count
                                            number of elements in buffer (integer)
19
20
       IN
                                            datatype of each buffer element (handle)
                datatype
21
       OUT
                status
                                            status object (Status)
22
23
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
^{24}
                    MPI_Datatype datatype, MPI_Status *status)
25
26
     MPI_File_write_all(fh, buf, count, datatype, status, ierror) BIND(C)
27
         TYPE(MPI_File), INTENT(IN) :: fh
28
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
^{29}
         INTEGER, INTENT(IN) :: count
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
         TYPE(MPI_Status) :: status
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
          <type> BUF(*)
35
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
36
37
         MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
38
     face.
39
40
41
42
43
44
45
46
47
48
```

MPI\_FILE\_IREAD(fh, buf, count, datatype, request) 1  $\mathbf{2}$ INOUT fh file handle (handle) 3 OUT buf initial address of buffer (choice) 4 IN count number of elements in buffer (integer) 56 IN datatype of each buffer element (handle) datatype 7 OUT request request object (handle) 8 9 int MPI\_File\_iread(MPI\_File fh, void \*buf, int count, 10 MPI\_Datatype datatype, MPI\_Request \*request) 11 12MPI\_File\_iread(fh, buf, count, datatype, request, ierror) BIND(C) 13 TYPE(MPI\_File), INTENT(IN) :: fh 14TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 15INTEGER, INTENT(IN) :: count 16TYPE(MPI\_Datatype), INTENT(IN) :: datatype 17TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI\_FILE\_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 20<type> BUF(\*) 21INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 22 23MPI\_FILE\_IREAD is a nonblocking version of the MPI\_FILE\_READ interface.  $^{24}$ 25**Example 13.3** The following Fortran code fragment illustrates file pointer update seman-26tics: 27Read the first twenty real words in a file into two local 28 ! 29 ! buffers. Note that when the first MPI\_FILE\_IREAD returns, 30 ! the file pointer has been updated to point to the 311 eleventh real word in the file. 32 33 integer bufsize, req1, req2 34 integer, dimension(MPI\_STATUS\_SIZE) :: status1, status2 35 parameter (bufsize=10) 36 buf1(bufsize), buf2(bufsize) real 37 integer (kind=MPI\_OFFSET\_KIND) zero 38 39 zero = 040 call MPI\_FILE\_OPEN( MPI\_COMM\_WORLD, 'myoldfile', & 41 MPI\_MODE\_RDONLY, MPI\_INFO\_NULL, myfh, ierr ) 42call MPI\_FILE\_SET\_VIEW( myfh, zero, MPI\_REAL, MPI\_REAL, 'native', & MPI\_INFO\_NULL, ierr ) 43 44call MPI\_FILE\_IREAD( myfh, buf1, bufsize, MPI\_REAL, & 45req1, ierr ) 46call MPI\_FILE\_IREAD( myfh, buf2, bufsize, MPI\_REAL, &

req2, ierr )

47

```
1
            call MPI_WAIT( req1, status1, ierr )
\mathbf{2}
            call MPI_WAIT( req2, status2, ierr )
3
4
            call MPI_FILE_CLOSE( myfh, ierr )
5
6
7
     MPI_FILE_IWRITE(fh, buf, count, datatype, request)
8
9
       INOUT
                 fh
                                            file handle (handle)
10
       IN
                 buf
                                            initial address of buffer (choice)
11
       IN
                count
                                            number of elements in buffer (integer)
12
                                            datatype of each buffer element (handle)
13
       IN
                datatype
14
       OUT
                 request
                                            request object (handle)
15
16
     int MPI_File_iwrite(MPI_File fh, const void *buf, int count,
17
                    MPI_Datatype datatype, MPI_Request *request)
18
19
     MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)
         TYPE(MPI_File), INTENT(IN) :: fh
20
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
         INTEGER, OPTIONAL, INTENT(OUT) ::
25
                                                 ierror
26
     MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
27
         <type> BUF(*)
28
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
29
30
         MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
^{31}
32
     MPI_FILE_SEEK(fh, offset, whence)
33
34
       INOUT
                fh
                                            file handle (handle)
35
       IN
                offset
                                            file offset (integer)
36
       IN
                whence
                                            update mode (state)
37
38
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
39
40
     MPI_File_seek(fh, offset, whence, ierror) BIND(C)
41
         TYPE(MPI_File), INTENT(IN) :: fh
42
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
43
         INTEGER, INTENT(IN) :: whence
44
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
45
46
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
47
          INTEGER FH, WHENCE, IERROR
48
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

	_FILE_SEEK updat possible values:	es the individual file pointer according to whence, which has the	1 $2$
• MP	$I\_SEEK\_SET:$ the po	pinter is set to offset	3 4
• MP	I_SEEK_CUR: the p	ointer is set to the current pointer position plus offset	5
• MP	I_SEEK_END: the p	ointer is set to the end of file plus offset	6 7
The	-	tive, which allows seeking backwards. It is erroneous to seek to	8 9 10 11
MPI_FILI	E_GET_POSITION(	(fh, offset)	12
IN	fh	file handle (handle)	13 14
OUT	offset	offset of individual pointer (integer)	15 16
int MPI	_File_get_positi	on(MPI_File fh, MPI_Offset *offset)	17 18
MPI_File	e_get_position(f	h, offset, ierror) BIND(C)	19
TYPE	E(MPI_File), INT	ENT(IN) :: fh	20
		FSET_KIND), INTENT(OUT) :: offset	21
INTE	EGER, OPTIONAL,	INTENT(OUT) :: ierror	22
MPI_FILE	E_GET_POSITION(F	H, OFFSET, IERROR)	23 24
	EGER FH, IERROR		24 25
INTE	EGER(KIND=MPI_OF	FSET_KIND) OFFSET	26
		ION returns, in offset, the current position of the individual file ve to the current view.	27 28
whe the MP	ence = MPI_SEEK_S current file pointer	e offset can be used in a future call to MPI_FILE_SEEK using ET to return to the current position. To set the displacement to position, first convert offset into an absolute byte position using E_OFFSET, then call MPI_FILE_SET_VIEW with the resulting f advice to users.)	29 30 31 32 33 34 35 36
MPI_FIL	E_GET_BYTE_OFF	SET(fh, offset, disp)	37
IN	fh	file handle (handle)	38 39
IN	offset	offset (integer)	40
OUT	disp	absolute byte position of offset (integer)	41
001	alsp	abbolate by the position of onset (integer)	42
int MPI_	_File_get_byte_o MPI_Offset	ffset(MPI_File fh, MPI_Offset offset, * *disp)	43 44 45
MDT E+1		-	45 46
	E_get_byte_onse E(MPI_File), INT	t(fh, offset, disp, ierror) BIND(C) ENT(IN) :: fh	47
		FSET_KIND), INTENT(IN) :: offset	48

12		ER(KIND=MPI_OFFSET_KIN ER, OPTIONAL, INTENT(C	-
3 4 5 6	INTEG	GET_BYTE_OFFSET(FH, OF ER FH, IERROR ER(KIND=MPI_OFFSET_KIN	
7 8 9 10	position. T		converts a view-relative offset into an absolute byte (from the beginning of the file) of <b>offset</b> relative to the
11	13.5.3 D	ata Access with Shared Fi	le Pointers
12 13 14 15 16 17 18 19	processes is the offset i update the nor update The sh	n the communicator group n the data access routines e shared file pointer mainta ed. nared file pointer routines h	pointer per collective MPI_FILE_OPEN (shared among ). The current value of this pointer implicitly specifies described in this section. These routines only use and ined by MPI. The individual file pointers are not used ave the same semantics as the data access with explicit 3.5.1 on page 507, with the following modifications:
20			urrent value of the MPI-maintained shared file pointer,
21 22 23	• the e		ared file pointer routines is defined to behave as if the
24 25 26	• the u file v	_	outines is erroneous unless all processes use the same
27 28 29 30 31 32	istic. The After point to th	user needs to use other syn a shared file pointer opera	ter routines, the serialization ordering is not determin- nchronization means to enforce a specific order. tion is initiated, the shared file pointer is updated to one that will be accessed. The file pointer is updated e.
33 34 35	Noncollecti	ve Operations	
36 37	MPI_FILE_	_READ_SHARED(fh, buf, c	ount, datatype, status)
38	INOUT	fh	file handle (handle)
39	OUT	buf	initial address of buffer (choice)
40 41	IN	count	number of elements in buffer (integer)
42	IN	datatype	datatype of each buffer element (handle)
43	OUT	status	status object (Status)
44 45 46	int MPI_F		le fh, void *buf, int count, type, MPI_Status *status)
47 48	MPI_File_		count, datatype, status, ierror) BIND(C)

	(MPI_File), INTENT(IN) ::		1 2
	(*), DIMENSION() :: b EER, INTENT(IN) :: count	uI	3
	(MPI_Datatype), INTENT(IN	) :: datatvpe	4
	(MPI_Status) :: status	51	5
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	6
MPT FTLE	READ SHARED (FH. BUF. COU	NT, DATATYPE, STATUS, IERROR)	7
	e> BUF(*)		8 9
• 1		STATUS(MPI_STATUS_SIZE), IERROR	9 10
MPI_I	FILE_READ_SHARED reads a	file using the shared file pointer.	11
_			12
	WDITE SUADED (the huf and	int datations status)	13
	_WRITE_SHARED(fh, buf, cou	, ,	14 15
INOUT	fh	file handle (handle)	15
IN	buf	initial address of buffer (choice)	17
IN	count	number of elements in buffer (integer)	18
IN	datatype	datatype of each buffer element (handle)	19
OUT	status	status object (Status)	20
001	514745	566663 05,000 (566662)	21
int MPI_H	File_write_shared(MPI_Fil	e fh, const void *buf, int count,	22 23
		e, MPI_Status *status)	24
MPT File	write shared (fh buf co	unt, datatype, status, ierror) BIND(C)	25
	(MPI_File), INTENT(IN) ::	• •	26
	(*), DIMENSION(), INTEN		27
INTEC	GER, INTENT(IN) :: count		28
	(MPI_Datatype), INTENT(IN	) :: datatype	29 30
	(MPI_Status) :: status	х	31
INTEC	GER, OPTIONAL, INTENT(OUT	) :: lerror	32
		UNT, DATATYPE, STATUS, IERROR)	33
• 1	<pre>&gt; BUF(*)</pre>		34
INTEC	ER FH, COUNT, DATATYPE,	STATUS(MPI_STATUS_SIZE), IERROR	35
MPI_I	FILE_WRITE_SHARED writes	a file using the shared file pointer.	36
			37 38
MPI_FILE	_IREAD_SHARED(fh, buf, cou	nt, datatype, request)	39
INOUT	fh	file handle (handle)	40
			41
OUT	buf	initial address of buffer (choice)	42
IN	count	number of elements in buffer (integer)	43
IN	datatype	datatype of each buffer element (handle)	44 45
OUT	request	request object (handle)	40
			47
			48

```
1
     int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
\mathbf{2}
                    MPI_Datatype datatype, MPI_Request *request)
3
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) BIND(C)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS ::
                                                       buf
6
         INTEGER, INTENT(IN) :: count
7
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                datatype
8
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
9
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
10
11
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
12
          <type> BUF(*)
13
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
14
         MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
15
     interface.
16
17
18
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
19
       INOUT
                fh
                                            file handle (handle)
20
       IN
                 buf
                                            initial address of buffer (choice)
21
22
       IN
                 count
                                            number of elements in buffer (integer)
23
       IN
                datatype
                                            datatype of each buffer element (handle)
^{24}
25
       OUT
                                            request object (handle)
                 request
26
27
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
28
                    MPI_Datatype datatype, MPI_Request *request)
29
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
30
         TYPE(MPI_File), INTENT(IN) :: fh
31
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
32
         INTEGER, INTENT(IN) :: count
33
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                datatype
34
         TYPE(MPI_Request), INTENT(OUT) ::
                                                 request
35
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
36
37
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
38
          <type> BUF(*)
39
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
40
         MPI_FILE_IWRITE_SHARED is a nonblocking version of the
41
     MPI_FILE_WRITE_SHARED interface.
42
43
     Collective Operations
44
45
     The semantics of a collective access using a shared file pointer is that the accesses to the
46
     file will be in the order determined by the ranks of the processes within the group. For each
47
     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.*)

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

	_READ_ORDERED(III, DUI, CO	unt, datatype, statusj	22
INOUT	fh	file handle (handle)	23
OUT	buf	initial address of buffer (choice)	24
IN	count	number of elements in buffer (integer)	25
IN	datatype	datatype of each buffer element (handle)	26
	datatype	datatype of each buller element (nandle)	27
OUT	status	status object (Status)	28
			29
int MPI_	File_read_ordered(MPI_File	e fh, void *buf, int count,	30
	MPI_Datatype datatyp	e, MPI_Status *status)	31
NDT 011			32
		<pre>int, datatype, status, ierror) BIND(C)</pre>	33
	(MPI_File), INTENT(IN) ::		34
	(*), DIMENSION() :: bu	11	35
	GER, INTENT(IN) :: count		36
	(MPI_Datatype), INTENT(IN)	) :: datatype	37
	(MPI_Status) :: status		38
INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror	39
MPI_FILE	_READ_ORDERED(FH, BUF, COU	JNT, DATATYPE, STATUS, IERROR)	40
	=		41
• 1		STATUS(MPI_STATUS_SIZE), IERROR	42
			43
	$FILE_READ_ORDERED$ is a co	ollective version of the MPI_FILE_READ_SHARED	44
interface.			45
			46
			47

MPI\_FILE\_READ\_ORDERED(fh, buf, count, datatype, status)

```
1
     MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       IN
                buf
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                                            datatype of each buffer element (handle)
                datatype
7
       OUT
                status
                                            status object (Status)
8
9
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
10
                    MPI_Datatype datatype, MPI_Status *status)
11
12
     MPI_File_write_ordered(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
     MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
20
          <type> BUF(*)
21
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
22
23
         MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
^{24}
     interface.
25
26
     Seek
27
28
     If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
29
     to call the following two routines (MPI_FILE_SEEK_SHARED and
30
     MPI_FILE_GET_POSITION_SHARED).
^{31}
32
     MPI_FILE_SEEK_SHARED(fh, offset, whence)
33
34
       INOUT
                fh
                                            file handle (handle)
35
       IN
                offset
                                            file offset (integer)
36
       IN
                whence
                                            update mode (state)
37
38
     int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
39
40
     MPI_File_seek_shared(fh, offset, whence, ierror) BIND(C)
41
         TYPE(MPI_File), INTENT(IN) :: fh
42
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
43
         INTEGER, INTENT(IN) :: whence
44
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
45
     MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
46
47
          INTEGER FH, WHENCE, IERROR
48
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

 $^{41}$ 

$MPI\_FILE\_SEEK\_SHARED$ updates the shared file pointer according to whence, we has the following possible values:	2
• MPI_SEEK_SET: the pointer is set to offset	3 4
$\bullet$ MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset	5
• MPI_SEEK_END: the pointer is set to the end of file plus offset	7
MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator grassociated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same variable for offset and whence. The offset can be negative, which allows seeking backwards. It is erroneous to see a negative position in the view.	lues 10 11 k to 12 13
MPI_FILE_GET_POSITION_SHARED(fh, offset)	14 15
IN fh file handle (handle)	16
OUT     offset     offset of shared pointer (integer)	17
onset of shared pointer (integer)	19
<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>	20 21
<pre>MPI_File_get_position_shared(fh, offset, ierror) BIND(C)     TYPE(MPI_File), INTENT(IN) :: fh     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	22 23 24 25
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	26 27 28 29
MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of shared file pointer in etype units relative to the current view.	the 30
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK_SHA using whence = MPI_SEEK_SET to return to the current position. To set the displement to the current file pointer position, first convert offset into an absolute position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW the resulting displacement. ( <i>End of advice to users.</i> )	lace- 34 byte 35

### 13.5.4 Split Collective Data Access Routines

MPI provides a restricted form of "nonblocking collective" I/O operations for all data accesses using split collective data access routines. These routines are referred to as "split" collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI\_FILE\_IREAD). The end routine completes the operation, much like the matching 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.

1 Split collective data access operations on a file handle fh are subject to the semantic  $\mathbf{2}$ rules given below. 3 4 • On any MPI process, each file handle may have at most one active split collective operation at any time. 56 • Begin calls are collective over the group of processes that participated in the collective 7 open and follow the ordering rules for collective calls. 8 9 • End calls are collective over the group of processes that participated in the collective 10 open and follow the ordering rules for collective calls. Each end call matches the 11 preceding begin call for the same collective operation. When an "end" call is made, 12exactly one unmatched "begin" call for the same operation must precede it. 13 • An implementation is free to implement any split collective data access routine using 14the corresponding blocking collective routine when either the begin call (e.g., 15MPI\_FILE\_READ\_ALL\_BEGIN) or the end call (e.g., MPI\_FILE\_READ\_ALL\_END) is 16issued. The begin and end calls are provided to allow the user and MPI implementation 17 to optimize the collective operation. 18 19 • Split collective operations do not match the corresponding regular collective opera-20tion. For example, in a single collective read operation, an MPI\_FILE\_READ\_ALL 21on one process does not match an MPI\_FILE\_READ\_ALL\_BEGIN/ 22 MPI\_FILE\_READ\_ALL\_END pair on another process. 2324• Split collective routines must specify a buffer in both the begin and end routines. 25By specifying the buffer that receives data in the end routine, we can avoid the 26problems described in "A Problem with Code Movements and Register Optimization," 27Section 17.1.17 on page 636, but not all of the problems described in Section 17.1.16 28on page 635. 29 • No collective I/O operations are permitted on a file handle concurrently with a split 30 collective access on that file handle (i.e., between the begin and end of the access). 31That is 32 33 34 MPI\_File\_read\_all\_begin(fh, ...); 35 . . . MPI\_File\_read\_all(fh, ...); 36 37 MPI\_File\_read\_all\_end(fh, ...); 38 39 is erroneous. 40 41 • In a multithreaded implementation, any split collective begin and end operation called 42by a process must be called from the same thread. This restriction is made to simplify 43 the implementation in the multithreaded case. (Note that we have already disallowed 44 having two threads begin a split collective operation on the same file handle since only 45one split collective operation can be active on a file handle at any time.) 46 4748

	0	e routines have the same meaning as for the equivalent collective	1
versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and			2
MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL).			3 4
	The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation		
		the matching end routine (i.e., MPI_FILE_READ_ALL_END)	5 6
-	produces the result as defined for the equivalent collective routine (i.e., MPI_FILE_READ_ALL).		
		sistency semantics (Section 13.8.1 on page 540), a matched pair	7 8
		ss operations (e.g., MPI_FILE_READ_ALL_BEGIN and	9
		)) compose a single data access.	10
		, I 0	11
MPL FILF	READ AT ALL	BEGIN(fh, offset, buf, count, datatype)	12
IN			13 14
		file handle (handle)	15
IN	offset	file offset (integer)	16
OUT	buf	initial address of buffer (choice)	17
IN	count	number of elements in buffer (integer)	18
IN	datatype	datatype of each buffer element (handle)	19
	51		20
int MPI_F	File_read_at_al	<pre>l_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	21 22
		, MPI_Datatype datatype)	22
MDT Eilo	road at all he	rin(th offset but count detetune ierror)	24
MPI_FIIe_	_read_at_all_be BIND(C)	gin(fh, offset, buf, count, datatype, ierror)	25
TYPE(MPI_File), INTENT(IN) :: fh			26
		FSET_KIND), INTENT(IN) :: offset	27
		), ASYNCHRONOUS :: buf	28
INTEG	ER, INTENT(IN)	:: count	29
TYPE(	(MPI_Datatype),	INTENT(IN) :: datatype	30
INTEG	ER, OPTIONAL,	INTENT(OUT) :: ierror	31
MPI FILE	READ AT ALL BE	GIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)	32 33
	e> BUF(*)		34
INTEG	ER FH, COUNT,	DATATYPE, IERROR	35
INTEG	ER(KIND=MPI_OF	FSET_KIND) OFFSET	36
			37
			38
MPI_FILE	_READ_AT_ALL_	END(fh, buf, status)	39
IN	fh	file handle (handle)	40
OUT	buf	initial address of buffer (choice)	41
			42
OUT	status	status object (Status)	43 44
int MDT T		1 and (MDI Eila the world whith MDI Chatter watater)	44 45
INT MPI_P	'ire_read_at_al	l_end(MPI_File fh, void *buf, MPI_Status *status)	46
		d(fh, buf, status, ierror) BIND(C)	47
TYPE(	TYPE(MPI_File), INTENT(IN) :: fh 48		

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
\mathbf{2}
         TYPE(MPI_Status) :: status
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
5
         <type> BUF(*)
6
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
7
8
9
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
10
11
       INOUT
                fh
                                            file handle (handle)
12
       IN
                offset
                                            file offset (integer)
13
                buf
       IN
                                            initial address of buffer (choice)
14
15
       IN
                count
                                            number of elements in buffer (integer)
16
       IN
                datatype
                                            datatype of each buffer element (handle)
17
18
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, const
19
                    void *buf, int count, MPI_Datatype datatype)
20
21
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
22
                    BIND(C)
23
         TYPE(MPI_File), INTENT(IN) :: fh
^{24}
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
26
         INTEGER, INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
30
          <type> BUF(*)
^{31}
         INTEGER FH, COUNT, DATATYPE, IERROR
32
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
33
34
35
     MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
36
37
       INOUT
                                            file handle (handle)
                fh
38
       IN
                buf
                                            initial address of buffer (choice)
39
       OUT
40
                status
                                            status object (Status)
41
42
     int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
43
                    MPI_Status *status)
44
     MPI_File_write_at_all_end(fh, buf, status, ierror) BIND(C)
45
         TYPE(MPI_File), INTENT(IN) :: fh
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
47
         TYPE(MPI_Status) :: status
48
```

INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	1
MPI_FILE_	MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)		
• 1	<type> BUF(*)</type>		
INTEG	ER FH, STATUS(MPI_STATUS_	SIZE), IERROR	5
			6
	READ_ALL_BEGIN(fh, buf, cc	(aquteteb tanu	7
	· ·		8 9
INOUT	fh	file handle (handle)	10
OUT	buf	initial address of buffer (choice)	11
IN	count	number of elements in buffer (integer)	12
IN	datatype	datatype of each buffer element (handle)	13 14
int MDT E	ile read all begin (MDT Fi	lo fhy world thuf int count	15
IIIC MFI_F	MPI_Datatype datatype	le fh, void *buf, int count,	16
NDT D'I			17
	read_all_begin(fh, buf, c MPI_File), INTENT(IN) ::	<pre>count, datatype, ierror) BIND(C) fh</pre>	18 19
	*), DIMENSION(), ASYNCH		20
	ER, INTENT(IN) :: count		21
	MPI_Datatype), INTENT(IN)	• =	22
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	23
MPI_FILE_	READ_ALL_BEGIN(FH, BUF, C	COUNT, DATATYPE, IERROR)	24 25
• -	> BUF(*)		25 26
INTEG	ER FH, COUNT, DATATYPE, I	ERROR	27
			28
	READ_ALL_END(fh, buf, stat	us)	29
	``	,	30
INOUT	fh	file handle (handle)	31 32
OUT	buf	initial address of buffer (choice)	33
OUT	status	status object (Status)	34
			35
int MPI_F	ile_read_all_end(MPI_File	e fh, void *buf, MPI_Status *status)	36
MPI_File_	read_all_end(fh, buf, sta	tus, ierror) BIND(C)	37 38
	<pre>MPI_File), INTENT(IN) ::</pre>		39
	*), DIMENSION(), ASYNCH MPI_Status) :: status	RONOUS :: buf	40
	ER, OPTIONAL, INTENT(OUT)	:: ierror	41
			42
	READ_ALL_END(FH, BUF, STA	TUS, 1ERROR)	43
• 1	> BUF(*) ER FH, STATUS(MPI_STATUS_	SIZE), TERROR	44 45
111100			46
			47
			48

```
1
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       IN
                buf
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
8
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
9
                    MPI_Datatype datatype)
10
11
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror) BIND(C)
12
          TYPE(MPI_File), INTENT(IN) :: fh
13
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
14
          INTEGER, INTENT(IN) :: count
15
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
17
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
18
          <type> BUF(*)
19
          INTEGER FH, COUNT, DATATYPE, IERROR
20
21
22
     MPI_FILE_WRITE_ALL_END(fh, buf, status)
23
^{24}
       INOUT
                fh
                                            file handle (handle)
25
       IN
                buf
                                            initial address of buffer (choice)
26
       OUT
                                            status object (Status)
                status
27
28
29
     int MPI_File_write_all_end(MPI_File fh, const void *buf,
30
                    MPI_Status *status)
31
     MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)
32
          TYPE(MPI_File), INTENT(IN) :: fh
33
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
34
          TYPE(MPI_Status) :: status
35
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
38
          <type> BUF(*)
39
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
40
41
42
43
44
45
46
47
48
```

MPI FILF	_READ_ORDERED_BEGIN(fh	. buf. count. datatype)	1
INOUT	fh	file handle (handle)	2
			3
OUT	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 7
			8
int MPI_H	0	PI_File fh, void *buf, int count,	9
	MPI_Datatype datatyp	pe)	10
MPI_File	_read_ordered_begin(fh, b	uf, count, datatype, ierror) BIND(C)	11
	(MPI_File), INTENT(IN) ::		12
	(*), DIMENSION(), ASYNC		13 14
	GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN		14
	GER, OPTIONAL, INTENT(OUT	• -	16
			17
	-	UF, COUNT, DATATYPE, IERROR)	18
01	e> BUF(*) GER FH, COUNT, DATATYPE,	TEBBUB	19
			20 21
			21
MPI_FILE	_READ_ORDERED_END(fh, b	ouf, status)	23
INOUT	fh	file handle (handle)	24
OUT	buf	initial address of buffer (choice)	25
			26
OUT	status	status object (Status)	27 28
int MDT I	File read ordered and (MDT	_File fh, void *buf, MPI_Status *status)	28 29
IIIC MFI_I	TIE_fead_ofdefed_end(MF1	_riie in, void *bui, Mri_Status *status)	30
		, status, ierror) BIND(C)	31
	(MPI_File), INTENT(IN) ::		32
	(*), DIMENSION(), ASYNC (MPI_Status) :: status	HRUNUUS :: DUI	33
	GER, OPTIONAL, INTENT(OUT	) :: ierror	34 35
MDT ETTE			36
	_READ_ORDERED_END(FH, BUF e> BUF(*)	, SIAIUS, IERRUR)	37
• 1	GER FH, STATUS(MPI_STATUS	_SIZE), IERROR	38
			39
			40
MPI_FILE	_WRITE_ORDERED_BEGIN(f	h, buf, count, datatype)	41 42
INOUT	fh	file handle (handle)	42
IN	buf	initial address of buffer (choice)	44
IN	count	number of elements in buffer (integer)	45
			46
IN	datatype	datatype of each buffer element (handle)	47
			48

```
1
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
\mathbf{2}
                   MPI_Datatype datatype)
3
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                  buf
6
         INTEGER, INTENT(IN) :: count
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                               ierror
9
10
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
11
         <type> BUF(*)
12
         INTEGER FH, COUNT, DATATYPE, IERROR
13
14
15
     MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
16
       INOUT
                                           file handle (handle)
                fh
17
18
       IN
                buf
                                           initial address of buffer (choice)
19
       OUT
                                           status object (Status)
                status
20
21
     int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
22
                   MPI_Status *status)
23
24
     MPI_File_write_ordered_end(fh, buf, status, ierror) BIND(C)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                  buf
27
         TYPE(MPI_Status) :: status
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
30
         <type> BUF(*)
^{31}
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
32
33
34
```

13.6 File Interoperability

35

At the most basic level, file interoperability is the ability to read the information previously written to a file — not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 13.7.2 on page 534) as well as the data conversion functions (Section 13.7.3 on page 535).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 13.8.1 on page 540), provided that it would have been possible to start the two processes simultaneously and have them reside 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.

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

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 13.7.3 on page 535). 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.*)

	,
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	
13	Rationale. This data representation allows the implementation to perform $I/O$
14	efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused ( <i>End of rationale</i> )
15	restrictions on how the file can be reused. ( <i>End of rationale.</i> )
16	Advice to implementors. Since "external32" is a superset of the functionality
17 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
23	
24	10 7
25	13.7
26	12.7.9 The data commission makes for communication also combe to these commissions
27	13.7.2. The data conversion rules for communication also apply to these conversions
28	(see Section 3.3.2 on page 35). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's
29	native representation.
30	•
31	This data representation has several advantages. First, all processes reading the file
32	in a heterogeneous MPI environment will automatically have the data converted to
33 34	their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that
35	the second environment will be able to read all the data in the file.
36	
37	The disadvantage of this data representation is that data precision and I/O perfor-
38	mance may be lost in data type conversions.
39	Advice to implementors. When implementing read and write operations on top
40	of MPI message-passing, the message data should be converted to and from the
41	"external32" representation in the client, and sent as type MPI_BYTE. This will
42	avoid possible double data type conversions and the associated further loss of
43	precision and performance. (End of advice to implementors.)
44	
45	13.7.1 Datatypes for File Interoperability
46	
47	If the file data representation is other than "native," care must be taken in constructing
48	etypes and filetypes. Any of the datatype constructor functions may be used; however,

for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4 on 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 and 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 routine MPI\_FILE\_GET\_TYPE\_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 using  $^{24}$ 

MPI\_TYPE\_CREATE\_RESIZED). This condition must also be fulfilled by any datatype that is used in the construction of the etype and filetype, if this datatype is replicated contiguously, either explicitly, by a call to MPI\_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 dependent.

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 on 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.*)

 $\mathbf{5}$ 

```
1
     MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)
2
       IN
                 fh
                                              file handle (handle)
3
       IN
                 datatype
                                              datatype (handle)
4
5
       OUT
                 extent
                                              datatype extent (integer)
6
\overline{7}
     int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
8
                     MPI Aint *extent)
9
     MPI_File_get_type_extent(fh, datatype, extent, ierror) BIND(C)
10
          TYPE(MPI_File), INTENT(IN) :: fh
11
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                   datatype
12
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
                                                                  extent
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
14
15
     MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
16
          INTEGER FH, DATATYPE, IERROR
17
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
18
          Returns the extent of datatype in the file fh. This extent will be the same for all
19
     processes accessing the file fh. If the current view uses a user-defined data representation
20
     (see Section 13.7.3 on page 535), MPI uses the dtype_file_extent_fn callback to calculate the
21
     extent.
22
23
           Advice to implementors. In the case of user-defined data representations, the extent of
24
           a derived datatype can be calculated by first determining the extents of the predefined
25
           datatypes in this derived datatype using dtype_file_extent_fn (see Section 13.7.3 on
26
           page 535). (End of advice to implementors.)
27
28
     13.7.2 External Data Representation: "external32"
29
30
     All MPI implementations are required to support the data representation defined in this
31
     section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required.
32
```

All floating point values are in big-endian IEEE format [37] of the appropriate size. 33 Floating point values are represented by one of three IEEE formats. These are the IEEE 34"Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, 35 respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 36 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous 37 to the "Double" format. All integral values are in two's complement big-endian format. Big-38 endian means most significant byte at lowest address byte. For C \_Bool, Fortran LOGICAL, 39 and C++ bool, 0 implies false and nonzero implies true. C float \_Complex, double 40\_Complex, and long double \_Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other 41 complex types are represented by a pair of floating point format values for the real and 42imaginary components. Characters are in ISO 8859-1 format [38]. Wide characters (of type 43MPI\_WCHAR) are in Unicode format [59].

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
 parts at the most significant bit of each part.

<sup>47</sup> According to IEEE specifications [37], the "NaN" (not a number) is system dependent.
 <sup>48</sup> It should not be interpreted within MPI as anything other than "NaN."

	-	PI treatment of "NaN" is similar to the approach used et/rfc/rfc1832.txt). ( <i>End of advice to implementors.</i> )	$\frac{1}{2}$	
All	All data is byte aligned, regardless of type. All data items are stored contiguously in			
the file (	the file (if the file view is contiguous).			
		tes of LOGICAL and bool must be checked to determine	6 7	
$^{\mathrm{the}}$	e value. (End of advice to imp	plementors.)	8	
Aa	<i>lvice to users.</i> The type M	PI_PACKED is treated as bytes and is not converted.	9	
Th		MPI_PACK has the option of placing a header in the	10 11	
	20 F (-		12 13	
	· ·	pes returned from MPI_TYPE_CREATE_F90_REAL, , and MPI_TYPE_CREATE_F90_INTEGER are defined	14	
in Sectio	on 17.1.9, page 624.		15 16	
4.0	lvice to implementors. Wh	en converting a larger size integer to a smaller size	17	
	-	it bytes are moved. Care must be taken to preserve	18	
	0, 0	no conversion errors if the data range is within the	19	
	0	. (End of advice to implementors.)	20	
			21	
Tab	le $13.2$ specifies the sizes of p	redefined datatypes in "external32" format.	22	
1070			23 24	
13.7.3	User-Defined Data Represen	tations	25	
There a	There are two situations that cannot be handled by the required representations:			
1. a ı	user wants to write a file in a	representation unknown to the implementation, and	27 28	
2. au	ser wants to read a file writte	n in a representation unknown to the implementation.	29	
Use	r-defined data representations	s allow the user to insert a third party converter into	30	
	stream to do the data represe	- · ·	31 32	
			33	
	CISTED DATABED (dataran	read_conversion_fn, write_conversion_fn,	34	
	dtype_file_extent_fn, e		35	
		,	36	
IN	datarep	data representation identifier (string)	37	
IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)	38 39	
IN	write_conversion_fn	function invoked to convert from native representation	40 41	
		to file representation (function)	42	
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as	43	
		represented in the file (function)	44	
IN	extra_state	extra state	45	
			46	
int MPI	_Register_datarep(const	-	47	
	MPT Dataren conver	sion function *read conversion fn	48	

MPI\_Datarep\_conversion\_function \*read\_conversion\_fn,

12	Туре	Length	Optional Type
3	MPI_PACKED	1	MPI_INTEGER1
4	MPI_BYTE	1	MPI_INTEGER2
5	MPI_CHAR	1	MPI_INTEGER4
6	MPI_UNSIGNED_CHAR	1	MPI_INTEGER8
7	MPI_SIGNED_CHAR	1	 MPI_INTEGER16
8	MPI_WCHAR	2	
9	MPI_SHORT	2	MPI_REAL2
10	MPI_UNSIGNED_SHORT	2	MPI_REAL4
11	MPI_INT	4	MPI_REAL8
12	MPI_UNSIGNED	4	MPI_REAL16
13	MPI_LONG	4	
14	MPI_UNSIGNED_LONG	4	MDI COMDIEXA
15			MPI_COMPLEX4
16	MPI_LONG_LONG_INT	8	MPI_COMPLEX8
17	MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16
18	MPI_FLOAT	4	MPI_COMPLEX32
19	MPI_DOUBLE	8	
20	MPI_LONG_DOUBLE	16	
21	MPI_C_BOOL	1	
22	MPI_INT8_T	1	C++ Types
23	MPI_INT16_T	2	
24	MPI_INT32_T	4	MPI_CXX_BOOL
25	MPI_INT64_T	4 8	
26			MPI_CXX_FLOAT_COMPLE
27	MPI_UINT8_T	1	MPI_CXX_DOUBLE_COMPI
28	MPI_UINT16_T	2	MPI_CXX_LONG_DOUBLE_
29	MPI_UINT32_T	4	
30	MPI_UINT64_T	8	
31	MPI_AINT	8	
32	MPI_COUNT	8	
33	MPI_OFFSET	8	
34	MPI_C_COMPLEX	2*4	
35	MPI_C_FLOAT_COMPLEX	2*4	
36	MPI_C_DOUBLE_COMPLEX	2*8	
37	MPI_C_LONG_DOUBLE_COMPLEX	2*16	
38			
39	MPI_CHARACTER	1	
40	MPI_LOGICAL	4	
41	MPI_INTEGER	4	
42	MPI_REAL	4	
43	MPI_DOUBLE_PRECISION	8	
44	MPI_COMPLEX	2*4	
45	MPI_DOUBLE_COMPLEX	2*8	
46			
47	Table 13.2:	"external32"	sizes of predefined datatypes
48	10.010 10.2.		is in provide data of pos

Optional Type	Length
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_INTEGER16	16
MPI_REAL2	2
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16
MPI_COMPLEX4	2*2
MPI_COMPLEX8	2*4
MPI_COMPLEX16	2*8
MPI_COMPLEX32	2*16

C++ Types Le	ength
MPI_CXX_BOOL	1
MPI_CXX_FLOAT_COMPLEX	2*4
MPI_CXX_DOUBLE_COMPLEX	2*8
MPI_CXX_LONG_DOUBLE_COMPLEX	2*16

```
MPI_Datarep_conversion_function *write_conversion_fn,
              MPI_Datarep_extent_function *dtype_file_extent_fn,
              void *extra_state)
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
              dtype_file_extent_fn, extra_state, ierror) BIND(C)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
    PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
              DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                      18
    INTEGER IERROR
                                                                                      19
    The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn
                                                                                      20
with the data representation identifier datarep. datarep can then be used as an argument
                                                                                      21
to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion
                                                                                      22
functions to convert all data items accessed between file data representation and native
                                                                                      23
representation. MPI_REGISTER_DATAREP is a local operation and only registers the data
                                                                                      24
representation for the calling MPI process. If datarep is already defined, an error in the
                                                                                      25
error class MPI_ERR_DUP_DATAREP is raised using the default file error handler (see Sec-
tion 13.9 on page 550). The length of a data representation string is limited to the value of
                                                                                      27
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;
```

```
Extent Callback
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
             MPI_Aint *file_extent, void *extra_state);
ABSTRACT INTERFACE
 SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
 ierror) BIND(C)
      TYPE(MPI_Datatype) :: datatype
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
      INTEGER :: ierror
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
   INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
```

it is not expected that an application will generate them in significant numbers.

The function dtype\_file\_extent\_fn must return, in file\_extent, the number of bytes re-46quired to store datatype in the file representation. The function is passed, in extra\_state, 47the argument that was passed to the MPI\_REGISTER\_DATAREP call. MPI will only call 48

1

2

3

4

5

6

7

9

10

11

1213

1415

16

17

26

28

29

30 31

32 33

34

35

36

37

38

39

40

41 42

43

44

```
1
     this routine with predefined datatypes employed by the user.
\mathbf{2}
3
     Datarep Conversion Functions
4
     typedef int MPI_Datarep_conversion_function(void *userbuf,
5
                     MPI_Datatype datatype, int count, void *filebuf,
6
                     MPI_Offset position, void *extra_state);
7
8
     ABSTRACT INTERFACE
9
        SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
10
        filebuf, position, extra_state, ierror) BIND(C)
11
            USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
12
            TYPE(C_PTR), VALUE :: userbuf, filebuf
13
            TYPE(MPI_Datatype) :: datatype
14
            INTEGER :: count, ierror
15
            INTEGER(KIND=MPI_OFFSET_KIND) :: position
16
            INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
17
     SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
18
                     POSITION, EXTRA_STATE, IERROR)
19
          <TYPE> USERBUF(*), FILEBUF(*)
20
          INTEGER COUNT, DATATYPE, IERROR
21
          INTEGER(KIND=MPI_OFFSET_KIND) POSITION
22
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
23
24
          The function read_conversion_fn must convert from file data representation to na-
25
     tive representation. Before calling this routine, MPI allocates and fills filebuf with count
26
     contiguous data items. The type of each data item matches the corresponding entry for the
27
     predefined datatype in the type signature of datatype. The function is passed, in extra_state,
28
     the argument that was passed to the MPI_REGISTER_DATAREP call. The function must
29
     copy all count data items from filebuf to userbuf in the distribution described by datatype,
30
     converting each data item from file representation to native representation. datatype will be
^{31}
     equivalent to the datatype that the user passed to the read function. If the size of datatype
32
     is less than the size of the count data items, the conversion function must treat datatype
33
     as being contiguously tiled over the userbuf. The conversion function must begin storing
34
     converted data at the location in userbuf specified by position into the (tiled) datatype.
35
36
           Advice to users. Although the conversion functions have similarities to MPI_PACK
37
           and MPI_UNPACK, one should note the differences in the use of the arguments count
38
           and position. In the conversion functions, count is a count of data items (i.e., count
39
           of typemap entries of datatype), and position is an index into this typemap. In
40
           MPI_PACK, incount refers to the number of whole datatypes, and position is a number
41
           of bytes. (End of advice to users.)
42
43
           Advice to implementors. A converted read operation could be implemented as follows:
44
             1. Get file extent of all data items
45
46
             2. Allocate a filebuf large enough to hold all count data items
47
             3. Read data from file into filebuf
48
```

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 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 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

33

34

35

36

37

38

39

40

41

42

43

44

45

write routines in Section 13.4 on page 504, 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.

<sup>5</sup> The conversion functions must be reentrant. User defined data representations are <sup>6</sup> restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion <sup>7</sup> 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.

10 11 12

13

8

9

# 13.7.4 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.

<sup>19</sup> Compatibility can be obtained when "external32" representation is used, although <sup>20</sup> precision may be lost and the performance may be less than when "native" representation is <sup>21</sup> used. Compatibility is guaranteed using "external32" provided at least one of the following <sup>22</sup> conditions is met.

 $23 \\ 24$ 

25

26 27

28

29

30

31

32 33

34

35 36

37

38

39 40 41

42

43

- The data access routines directly use types enumerated in Section 13.7.2 on page 534, 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 17.1.9 on page 620).
- 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 compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 17.1.9 on page 620 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

# 13.8 Consistency and Semantics

13.8.1 File Consistency

<sup>44</sup> <sup>45</sup> Consistency semantics define the outcome of multiple accesses to a single file. All file <sup>46</sup> accesses in MPI are relative to a specific file handle created from a collective open. MPI <sup>47</sup> provides three levels of consistency: sequential consistency among all accesses using a single <sup>48</sup> 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  $FH_2$  may be different, the groups of processes used for each open may or may not intersect, the file handles in  $FH_1$  may be destroyed before those in  $FH_2$  are created, etc. Consider the following three cases: a single file handle (e.g.,  $fh_1 \in FH_1$ ), two file handles created from 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 13.5.4 on page 523) 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 non-blocking 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).

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.

Case 1:  $fh_1 \in FH_1$  All operations on  $fh_1$  are sequentially consistent if atomic mode is 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.

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

 $\mathbf{6}$ 

 $\overline{7}$ 

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

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.

```
<sup>6</sup> Case 3: fh_1 \in FH_1 and fh_2 \in FH_2 Consider access to a single file using file handles from

<sup>7</sup> distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC

<sup>8</sup> must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).
```

<sup>9</sup> Sequential consistency is guaranteed among accesses to a single file if for any write <sup>10</sup> sequence  $SEQ_1$  to the file, there is no sequence  $SEQ_2$  to the file which is *concurrent* with <sup>11</sup>  $SEQ_1$ . To guarantee sequential consistency when there are write sequences,

<sup>12</sup> MPI\_FILE\_SYNC must be used together with a mechanism that guarantees nonconcurrency
 <sup>13</sup> of the sequences.

See the examples in Section 13.8.10 on page 546 for further clarification of some of these consistency semantics.

<sup>18</sup> MPI\_FILE\_SET\_ATOMICITY(fh, flag)

19	INOUT	fh	file handle (handle)				
20 21	IN	flag	true to set atomic mode, false to set nonatomic mode				
21		0	(logical)				
23							
24	<pre>int MPI_File_set_atomicity(MPI_File fh, int flag)</pre>						
25	MPI_File_set_atomicity(fh, flag, ierror) BIND(C)						
26	TYPE(MPI_File), INTENT(IN) :: fh						
27 28	LOGICAL, INTENT(IN) :: flag						
29	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror				
30	MPI_FILE_	SET_ATOMICITY(FH, FLAG, 1	IERROR)				
31	INTEG	ER FH, IERROR					
32	LOGIC	CAL FLAG					
33	T -+ E						

Let FH be the set of file handles created by one collective open. The consistency semantics for data access operations using FH is set by collectively calling

<sup>35</sup> MPI\_FILE\_SET\_ATOMICITY on *FH*. MPI\_FILE\_SET\_ATOMICITY is collective; all pro-<sup>36</sup> cesses in the group must pass identical values for fh and flag. If flag is true, atomic mode is <sup>37</sup> set; if flag is false, nonatomic mode is set.

<sup>38</sup> Changing the consistency semantics for an open file only affects new data accesses. <sup>39</sup> All completed data accesses are guaranteed to abide by the consistency semantics in effect <sup>40</sup> during their execution. Nonblocking data accesses and split collective operations that have <sup>41</sup> not completed (e.g., via MPI\_WAIT) are only guaranteed to abide by nonatomic mode <sup>42</sup> 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.)

1

 $^{2}$ 

3

4

 $\mathbf{5}$ 

14

15

16 17

MPI_FILE_GET_ATOMICITY(fh, flag) <sup>1</sup>				
	,		2	
IN	fh	file handle (handle)	3	
OUT	flag	true if atomic mode, false if nonatomic mode (logical)	4	
			5	
int MPI_H	File_get_atomicity(MPI_Fil	le fh, int *flag)	6	
MPT Filo	_get_atomicity(fh, flag, :	ierror) BIND(C)	7	
	(MPI_File), INTENT(IN) ::	fh	8	
	CAL, INTENT(OUT) :: flag		9	
	GER, OPTIONAL, INTENT(OUT)	) :: ierror	10	
MDT ETTE			11 12	
	_GET_ATOMICITY(FH, FLAG, 1 GER FH, IERROR	LERROR)	12	
	CAL FLAG		14	
			15	
		as the current consistency semantics for data access	16	
-		ated by one collective open. If flag is true, atomic	17	
mode is er	nabled; if flag is false, nonatom	1c mode 1s enabled.	18	
			19	
MPI_FILE	_SYNC(fh)		20	
INOUT	fh	file handle (handle)	21	
			22 23	
int MPI H	File_sync(MPI_File fh)		23	
	-		25	
	_sync(fh, ierror) BIND(C)	<u>61</u>	26	
	(MPI_File), INTENT(IN) :: GER, OPTIONAL, INTENT(OUT)	fh ) :: ierror	27	
	ER, OFFICIAL, INTENT(001)		28	
	SYNC(FH, IERROR)		29	
INTE	GER FH, IERROR		30	
Callin	$_{ m MPI_FILE\_SYNC}$ with fh ca	uses all previous writes to fh by the calling process	31	
to be trans	sferred to the storage device. I	f other processes have made updates to the storage	32 33	
device, the	en all such updates become visi	ble to subsequent reads of <b>fh</b> by the calling process.	34	
	_SYNC may be necessary to e	ensure sequential consistency in certain cases (see	35	
above).			36	
	FILE_SYNC is a collective oper		37	
	-	g that all nonblocking requests and split collective fore calling MPI_FILE_SYNC — otherwise, the call	38	
*	LE_SYNC is erroneous.	sole cannig with_file_3 five — otherwise, the can	39	
00 MII 1_1 1			40	
13.8.2 R	andom Access vs. Sequential	Files	41	
			42	
		ess files from sequential stream files, such as pipes	43	
-	_	nust be opened with the MPI_MODE_SEQUENTIAL	44 45	
-		e only permitted data access operations are shared and etypes with holes are erroneous. In addition, the	45 46	
-		therefore calls to MDI EILE SEEK SHAPED and	47	

notion of file pointer is not meaningful; therefore, calls to  $\mathsf{MPI}_\mathsf{FILE}_\mathsf{SEEK}_\mathsf{SHARED}$  and

MPI\_FILE\_GET\_POSITION\_SHARED are erroneous, and the pointer update rules specified

47

for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

*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.

13.8.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on imple mentors. In cases where the progress rules restrict possible implementation choices more
 than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such
 as resource exhaustion) causes an error.

<sup>19</sup> Nonblocking data access routines inherit the following progress rule from nonblocking
 <sup>20</sup> point to point communication: a nonblocking write is equivalent to a nonblocking send for
 <sup>21</sup> which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking
 <sup>22</sup> 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.

27 28

37

38

## 13.8.4 Collective File Operations

<sup>29</sup> Collective file operations are subject to the same restrictions as collective communication <sup>31</sup> operations. For a complete discussion, please refer to the semantics set forth in Section 5.13 <sup>32</sup> on page 214.

Collective file operations are collective over a duplicate 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.

13.8.5 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.*)

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11 12

#### 13.8.6 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.

### 13.8.7 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 with kind parameter MPI\_OFFSET\_KIND, which is defined in the mpi\_f08 module, the mpi module and the deprecated 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 17.2 on page 649).

#### 13.8.8 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 portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as *hints* specified via info when a file is created (see Section 13.2.8 on page 498).

### 13.8.9 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.

<sup>19</sup> ticketWG.

 $^{31}$ 

1 When applying consistency semantics, calls to MPI\_FILE\_SET\_SIZE and  $\mathbf{2}$ MPI\_FILE\_PREALLOCATE are considered writes to the file (which conflict with operations 3 that access bytes at displacements between the old and new file sizes), and 4 MPI\_FILE\_GET\_SIZE is considered a read of the file (which overlaps with all accesses to the 5file). 6 Advice to users. Any sequence of operations containing the collective routines 7 MPI\_FILE\_SET\_SIZE and MPI\_FILE\_PREALLOCATE is a write sequence. As such, 8 sequential consistency in nonatomic mode is not guaranteed unless the conditions in 9 Section 13.8.1 on page 540 are satisfied. (End of advice to users.) 10 11 File pointer update semantics (i.e., file pointers are updated by the amount accessed) 12are only guaranteed if file size changes are sequentially consistent. 13 14Consider the following example. Given two operations made by Advice to users. 15separate processes to a file containing 100 bytes: an MPI\_FILE\_READ of 10 bytes and 16an MPI\_FILE\_SET\_SIZE to 0 bytes. If the user does not enforce sequential consis-17 tency between these two operations, the file pointer may be updated by the amount 18 requested (10 bytes) even if the amount accessed is zero bytes. (End of advice to 19users.) 202113.8.10 Examples 22 23The examples in this section illustrate the application of the MPI consistency and semantics  $^{24}$ guarantees. These address 2526• conflicting accesses on file handles obtained from a single collective open, and 27• all accesses on file handles obtained from two separate collective opens. 2829 The simplest way to achieve consistency for conflicting accesses is to obtain sequential 30 consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10  $^{31}$ integers. If the latter, every element of b will be 5. If nonatomic mode is set, the results of 32 the read are undefined. 33 34/\* Process 0 \*/ 35 int i, a[10] ; TRUE = 1; 36 int 37 38for ( i=0;i<10;i++)</pre> 39 a[i] = 5;40 $^{41}$ MPI\_File\_open( MPI\_COMM\_WORLD, "workfile", 42MPI\_MODE\_RDWR | MPI\_MODE\_CREATE, MPI\_INFO\_NULL, &fh0 ); MPI\_File\_set\_view( fh0, 0, MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL ) ; 4344MPI\_File\_set\_atomicity( fh0, TRUE ) ; 45MPI\_File\_write\_at(fh0, 0, a, 10, MPI\_INT, &status) ; 46/\* MPI\_Barrier( MPI\_COMM\_WORLD ) ; \*/ 4748

```
1
/* Process 1 */
                                                                                         2
int b[10];
                                                                                         3
int TRUE = 1;
MPI_File_open( MPI_COMM_WORLD, "workfile",
                                                                                         4
                 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
                                                                                         5
                                                                                         6
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                         7
MPI_File_set_atomicity( fh1, TRUE ) ;
                                                                                         8
/* MPI_Barrier( MPI_COMM_WORLD ) ; */
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status) ;
                                                                                         9
                                                                                        10
A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
                                                                                        11
temporal order with, for example, calls to MPI_BARRIER.
                                                                                        12
                                                                                        13
     Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
                                                                                        14
     order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
                                                                                        15
     received by process 1 using MPI_RECV. (End of advice to users.)
                                                                                        16
                                                                                        17
    Alternatively, a user can impose consistency with nonatomic mode set:
                                                                                        18
/* Process 0 */
                                                                                        19
int i, a[10] ;
                                                                                        20
for ( i=0;i<10;i++)
                                                                                        21
   a[i] = 5;
                                                                                        22
                                                                                        23
MPI_File_open( MPI_COMM_WORLD, "workfile",
                                                                                        ^{24}
                 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
                                                                                        25
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                        26
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
                                                                                        27
MPI_File_sync( fh0 ) ;
                                                                                        28
MPI_Barrier( MPI_COMM_WORLD ) ;
                                                                                        29
MPI_File_sync( fh0 ) ;
                                                                                        30
                                                                                        31
/* Process 1 */
                                                                                        32
int b[10];
                                                                                        33
MPI_File_open( MPI_COMM_WORLD, "workfile",
                                                                                        34
                 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
                                                                                        35
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                        36
MPI_File_sync( fh1 ) ;
                                                                                        37
MPI_Barrier( MPI_COMM_WORLD ) ;
                                                                                        38
MPI_File_sync( fh1 ) ;
                                                                                        39
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
                                                                                        40
                                                                                        41
The "sync-barrier-sync" construct is required because:
                                                                                        42
   • The barrier ensures that the write on process 0 occurs before the read on process 1.
                                                                                        43
                                                                                        44
   • The first sync guarantees that the data written by all processes is transferred to the
                                                                                        45
     storage device.
                                                                                        46
                                                                                        47
   • The second sync guarantees that all data which has been transferred to the storage
                                                                                        48
     device is visible to all processes. (This does not affect process 0 in this example.)
```

1

 $\mathbf{2}$ 

The following program represents an erroneous attempt to achieve consistency by eliminating the apparently superfluous second "sync" call for each process.

```
3
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
4
     /* Process 0 */
5
     int i, a[10] ;
6
     for ( i=0;i<10;i++)</pre>
7
        a[i] = 5;
8
9
     MPI_File_open( MPI_COMM_WORLD, "workfile",
10
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
11
     MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
12
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
13
     MPI_File_sync( fh0 ) ;
14
     MPI_Barrier( MPI_COMM_WORLD ) ;
15
16
     /* Process 1 */
17
     int b[10] ;
18
     MPI_File_open( MPI_COMM_WORLD, "workfile",
19
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
20
     MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
21
     MPI_Barrier( MPI_COMM_WORLD ) ;
22
     MPI_File_sync( fh1 ) ;
23
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
24
25
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
26
27
     The above program also violates the MPI rule against out-of-order collective operations and
28
     will deadlock for implementations in which MPI_FILE_SYNC blocks.
29
30
          Advice to users. Some implementations may choose to implement MPI_FILE_SYNC
31
          as a temporally synchronizing function. When using such an implementation, the
32
          "sync-barrier-sync" construct above can be replaced by a single "sync." The results of
33
          using such code with an implementation for which MPI_FILE_SYNC is not temporally
34
          synchronizing is undefined. (End of advice to users.)
35
36
     Asynchronous I/O
37
38
     The behavior of asynchronous I/O operations is determined by applying the rules specified
     above for synchronous I/O operations.
39
         The following examples all access a preexisting file "myfile." Word 10 in myfile initially
40
41
     contains the integer 2. Each example writes and reads word 10.
42
         First consider the following code fragment:
43
     int a = 4, b, TRUE=1;
44
     MPI_File_open( MPI_COMM_WORLD, "myfile",
45
                     MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
46
     MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
47
     /* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
48
```

```
1
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
                                                                                         \mathbf{2}
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &regs[1]);
                                                                                         3
MPI_Waitall(2, reqs, statuses) ;
                                                                                        4
For asynchronous data access operations, MPI specifies that the access occurs at any time
                                                                                         5
between the call to the asynchronous data access routine and the return from the corre-
                                                                                         6
sponding request complete routine. Thus, executing either the read before the write, or the
                                                                                        7
write before the read is consistent with program order. If atomic mode is set, then MPI
                                                                                         8
guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic
                                                                                         9
mode is not set, then sequential consistency is not guaranteed and the program may read
                                                                                        10
something other than 2 or 4 due to the conflicting data access.
                                                                                        11
    Similarly, the following code fragment does not order file accesses:
                                                                                        12
                                                                                        13
int a = 4, b;
                                                                                        14
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                        15
                 MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                        16
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                        17
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
                                                                                        18
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
                                                                                        19
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
                                                                                        20
MPI_Wait(&reqs[0], &status) ;
                                                                                        21
MPI_Wait(&regs[1], &status) ;
                                                                                        22
If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee
                                                                                        23
                                                                                        ^{24}
sequential consistency in nonatomic mode.
                                                                                        25
    On the other hand, the following code fragment:
                                                                                        26
int a = 4, b;
                                                                                        27
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                        28
                 MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                        29
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                        30
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
                                                                                        31
MPI_Wait(&reqs[0], &status) ;
                                                                                        32
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
                                                                                        33
MPI_Wait(&reqs[1], &status) ;
                                                                                        34
                                                                                        35
defines the same ordering as:
                                                                                        36
                                                                                        37
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                        38
                 MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                        39
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                        40
                                                                                        41
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status );
                                                                                        42
MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status );
                                                                                        43
Since
                                                                                        44
                                                                                        45
   • nonconcurrent operations on a single file handle are sequentially consistent, and
                                                                                        46
   • the program fragments specify an order for the operations,
                                                                                        47
                                                                                        48
```

<sup>1</sup> MPI guarantees that both program fragments will read the value 4 into b. There is no need <sup>2</sup> to set atomic mode for this example.

Similar considerations apply to conflicting accesses of the form:

```
4
5 MPI_File_write_all_begin(fh,...);
6 MPI_File_iread(fh,...);
7 MPI_Wait(fh,...);
```

8 MPI\_File\_write\_all\_end(fh,...);

Recall that constraints governing consistency and semantics are not relevant to the following:

```
<sup>12</sup> MPI_File_write_all_begin(fh,...);
<sup>13</sup> MPI_File_read_all_begin(fh,...);
<sup>14</sup> MPI_File_read_all_end(fh,...);
<sup>15</sup> MPI_File_write_all_end(fh,...);
```

since split collective operations on the same file handle may not overlap (see Section 13.5.4 on page 523).

19 20

21

26

27

28

29

30 31

# 13.9 I/O Error Handling

<sup>22</sup> By default, communication errors are fatal — MPI\_ERRORS\_ARE\_FATAL is the default error
<sup>23</sup> handler associated with MPI\_COMM\_WORLD. I/O errors are usually less catastrophic (e.g.,
<sup>24</sup> "file not found") than communication errors, and common practice is to catch these errors
<sup>25</sup> 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 on page 340.

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 MPI\_FILE\_OPEN or MPI\_FILE\_DELETE), the first argument passed to the error handler is MPI\_FILE\_NULL.

I/O error handling differs from communication error handling in another important 39 aspect. By default, the predefined error handler for file handles is MPI\_ERRORS\_RETURN. 40The default file error handler has two purposes: when a new file handle is created (by 41 MPI\_FILE\_OPEN), the error handler for the new file handle is initially set to the default 42error handler, and I/O routines that have no valid file handle on which to raise an error 43 (e.g., MPI\_FILE\_OPEN or MPI\_FILE\_DELETE) use the default file error handler. The de-44fault file error handler can be changed by specifying MPI\_FILE\_NULL as the fh argument 45to MPI\_FILE\_SET\_ERRHANDLER. The current value of the default file error handler can 46 be determined by passing MPI\_FILE\_NULL as the fh argument to 47MPI\_FILE\_GET\_ERRHANDLER. 48

3

9

10

*Rationale.* For communication, the default error handler is inherited from 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.*)

# 13.10 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.

MPI_ERR_FILE	Invalid file handle	14
MPI_ERR_NOT_SAME	Collective argument not identical on all	15
	processes, or collective routines called in	16
	a different order by different processes	17
MPI_ERR_AMODE	Error related to the amode passed to	18
	MPI_FILE_OPEN	19
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	20
	MPI_FILE_SET_VIEW	21
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	22
	a file which supports sequential access only	23
MPI_ERR_NO_SUCH_FILE	File does not exist	24
MPI_ERR_FILE_EXISTS	File exists	25
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	26
MPI_ERR_ACCESS	Permission denied	27
MPI_ERR_NO_SPACE	Not enough space	28
MPI_ERR_QUOTA	Quota exceeded	29
MPI_ERR_READ_ONLY	Read-only file or file system	30
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	31
	the file is currently open by some process	32
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	33
	tered because a data representation identi-	34
	fier that was already defined was passed to	35
	MPI_REGISTER_DATAREP	36
MPI_ERR_CONVERSION	An error occurred in a user supplied data	37
	conversion function.	38
MPI_ERR_IO	Other I/O error	39
		40
Table 13.3	3: I/O Error Classes	41

 $\mathbf{2}$ 

```
13.11 Examples
1
\mathbf{2}
    13.11.1 Double Buffering with Split Collective I/O
3
4
    This example shows how to overlap computation and output. The computation is performed
5
    by the function compute_buffer().
6
7
    8
     *
9
     * Function:
                           double_buffer
10
     *
11
     * Synopsis:
12
          void double_buffer(
     *
13
                   MPI_File fh,
                                                         ** IN
     *
14
                   MPI_Datatype buftype,
     *
                                                         ** IN
15
                   int bufcount
                                                         ** IN
     *
16
     *
            )
17
     *
^{18}
     * Description:
19
            Performs the steps to overlap computation with a collective write
     *
20
            by using a double-buffering technique.
     *
21
     *
22
     * Parameters:
23
            fh
                              previously opened MPI file handle
     *
^{24}
                            MPI datatype for memory layout
            buftype
     *
25
     *
                              (Assumes a compatible view has been set on fh)
26
                             # buftype elements to transfer
     *
            bufcount
27
     *-----*/
28
29
    /* this macro switches which buffer "x" is pointing to */
30
    #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
^{31}
32
    void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount)
33
    {
34
       MPI_Status status; /* status for MPI calls */
35
36
       float *buffer1, *buffer2; /* buffers to hold results */
37
       float *compute_buf_ptr; /* destination buffer */
38
                               /* for computing */
       float *write_buf_ptr; /* source for writing */
39
40
                               /* determines when to quit */
       int done;
41
42
       /* buffer initialization */
43
       buffer1 = (float *)
44
                         malloc(bufcount*sizeof(float)) ;
45
       buffer2 = (float *)
46
                         malloc(bufcount*sizeof(float)) ;
47
       compute_buf_ptr = buffer1 ; /* initially point to buffer1 */
48
       write_buf_ptr = buffer1 ; /* initially point to buffer1 */
```

```
\mathbf{2}
                                                                                   3
/* DOUBLE-BUFFER prolog:
     compute buffer1; then initiate writing buffer1 to disk
                                                                                   4
 *
 */
                                                                                   5
                                                                                   6
compute_buffer(compute_buf_ptr, bufcount, &done);
                                                                                   7
MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
                                                                                   8
                                                                                   9
/* DOUBLE-BUFFER steady state:
                                                                                  10
    Overlap writing old results from buffer pointed to by write_buf_ptr
 *
                                                                                  11
 *
    with computing new results into buffer pointed to by compute_buf_ptr.
 *
                                                                                  12
    There is always one write-buffer and one compute-buffer in use
                                                                                  13
 *
    during steady state.
                                                                                  14
 *
                                                                                  15
 */
                                                                                  16
while (!done) {
                                                                                  17
   TOGGLE_PTR(compute_buf_ptr);
                                                                                  18
   compute_buffer(compute_buf_ptr, bufcount, &done);
                                                                                  19
   MPI_File_write_all_end(fh, write_buf_ptr, &status);
   TOGGLE_PTR(write_buf_ptr);
                                                                                  20
                                                                                  21
   MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
}
                                                                                  22
                                                                                  23
                                                                                  24
/* DOUBLE-BUFFER epilog:
                                                                                  25
 *
     wait for final write to complete.
                                                                                  26
 */
MPI_File_write_all_end(fh, write_buf_ptr, &status);
                                                                                  27
                                                                                  28
                                                                                  29
                                                                                  30
/* buffer cleanup */
                                                                                  31
free(buffer1);
                                                                                  32
free(buffer2);
                                                                                  33
                                                                                  34
                                                                                  35
```

# 13.11.2 Subarray Filetype Constructor

Assume we are writing out a 100x100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 13.4). To create the filetypes for each process one could use the following C program (see

# 13.12

}

```
4.1.3):
```

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
```

1

36

37

38

39

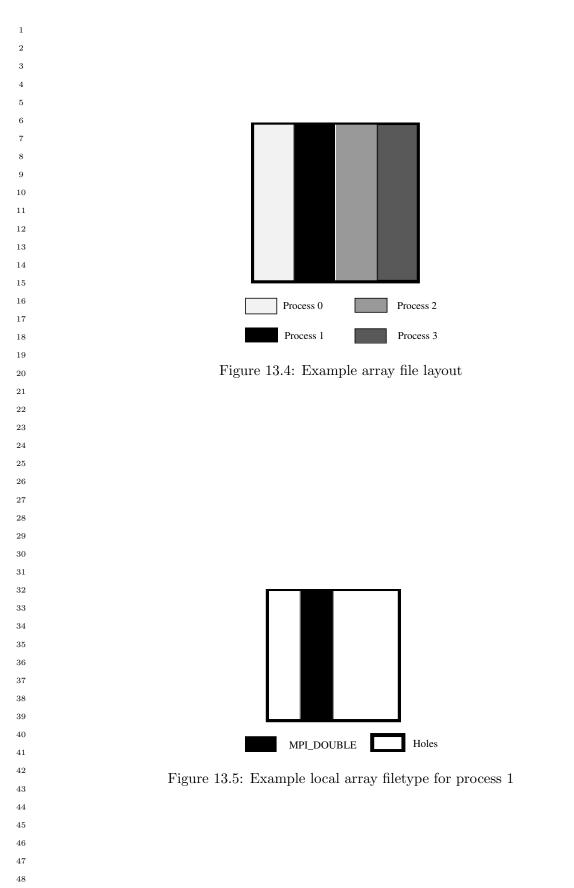
40 41 42

43 44

45

46

47



```
1
int rank;
                                                                                    \mathbf{2}
                                                                                    3
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
                                                                                    4
subsizes[0]=100; subsizes[1]=25;
                                                                                    5
starts[0]=0; starts[1]=rank*subsizes[1];
                                                                                    6
                                                                                    7
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                                                                                    8
                           MPI_DOUBLE, &filetype);
                                                                                    9
                                                                                   10
 Or, equivalently in Fortran:
                                                                                   11
                                                                                   12
    double precision subarray(100,25)
                                                                                   13
    integer filetype, rank, ierror
                                                                                   14
    integer sizes(2), subsizes(2), starts(2)
                                                                                   15
                                                                                   16
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
                                                                                   17
    sizes(1)=100
                                                                                   18
    sizes(2)=100
                                                                                   19
    subsizes(1)=100
                                                                                   20
    subsizes(2)=25
                                                                                   21
    starts(1)=0
                                                                                   22
    starts(2)=rank*subsizes(2)
                                                                                   23
                                                                                   ^{24}
    call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
                                                                                   25
                MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                    &
                                                                                   26
                filetype, ierror)
                                                                                   27
                                                                                   28
```

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.

# Chapter 14

# **Tool Support**

# 14.1 Introduction

This chapter discusses interfaces that allow 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.

# 14.2 Profiling Interface

# 14.2.1 Requirements

To meet the requirements for the  $\mathsf{MPI}$  profiling interface, an implementation of the  $\mathsf{MPI}$  functions must

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI\_ for each MPI function in each provided language binding and language support method. For routines 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 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 17.1.5 on page 609.

- 2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
- 3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of "wrapper" functions that call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

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.

 $11 \\ 12$ 

1

2

3

4

5

6

7

8

9

10

13 14

14.2.2 Discussion

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.

5. provide a no-op routine MPI\_PCONTROL in the MPI library.

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 executable 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 section is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others that would be equally valid).

44 45

# 14.2.3 Logic of the Design

<sup>46</sup> Provided that an MPI implementation meets the requirements above, it is possible for <sup>47</sup> the implementor of the profiling system to intercept the MPI calls that are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

# 14.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of MPI\_PCONTROL.

MPI\_PCONTROL(level, ...)

IN	level			Profiling level (integer)
int	MPI_Pcontrol(const	int	level,	)

MPI\_Pcontrol(level) BIND(C)
INTEGER, INTENT(IN) :: level

```
MPI_PCONTROL(LEVEL)
INTEGER LEVEL
```

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to MPI\_PCONTROL. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of level.

level==0 Profiling is disabled.
level==1 Profiling is enabled at a normal default level of detail.
level==2 Profile buffers are flushed, which may be a no-op in some profilers.
All other values of level have profile library defined effects and additional arguments.

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 with the argument 1). This allows users to link with a profiling library and to obtain profile output without having to modify their source code at all.

The provision of MPI\_PCONTROL as a no-op in the standard MPI library supports the collection of more detailed profiling information with source code that can still link against the standard MPI library. 48

 $\mathbf{2}$ 

# 14.2.5 Profiler Implementation Example

A profiler can accumulate the total amount of data sent by the MPI\_SEND function, along with the total elapsed time spent in the function as the following example shows:

```
\mathbf{5}
     Example 14.1
6
7
     static int totalBytes = 0;
     static double totalTime = 0.0;
8
9
10
     int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
11
                    int dest, int tag, MPI_Comm comm)
12
     {
                                                /* Pass on all arguments */
13
         double tstart = MPI_Wtime();
14
         int extent;
15
                         = PMPI_Send(buffer,count,datatype,dest,tag,comm);
         int result
16
17
         totalTime += MPI_Wtime() - tstart;
                                                           /* and time
                                                                                   */
18
19
         MPI_Type_size(datatype, &extent); /* Compute size */
20
         totalBytes += count*extent;
21
22
         return result;
23
     }
^{24}
25
             MPI Library Implementation Example
     14.2.6
26
     If the MPI library is implemented in C on a Unix system, then there are various options,
27
     including the two presented here, for supporting the name-shift requirement. The choice
28
     between these two options depends partly on whether the linker and compiler support weak
29
     symbols.
30
^{31}
     Systems with Weak Symbols
32
33
     If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System
34
     V.4 machines), then only a single library is required as the following example shows:
35
36
     Example 14.2
37
     #pragma weak MPI_Example = PMPI_Example
38
39
     int PMPI_Example(/* appropriate args */)
40
     {
41
          /* Useful content */
42
     }
43
44
          The effect of this #pragma is to define the external symbol MPI_Example as a weak
```

<sup>45</sup> definition. This means that the linker will not complain if there is another definition of the <sup>46</sup> symbol (for instance in the profiling library); however if no other definition exists, then the <sup>47</sup> linker will use the weak definition.

1

 $\mathbf{2}$ 

3

# Systems Without Weak Symbols

In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

# Example 14.3

```
#ifdef PROFILELIB
# ifdef __STDC__
# define FUNCTION(name) P##name
# else
# define FUNCTION(name) P/**/name
# endif
#else
# define FUNCTION(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
```

/\* Useful content \*/
}

The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILELIB macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

# % cc ... -lmyprof -lpmpi -lmpi

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the "name shifted" 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 func-tions (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., it might allow one to answer the question "How much time is spent in the point to point routines when they are called from collective functions?"), we have decided not to enforce any restrictions on the author of the MPI library that would

 $\frac{4}{5}$ 

 $^{24}$ 

overcome this. Therefore the author of the profiling library should be aware of this problem,
 and guard against it. 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

6 7

The Unix linker traditionally operates in one 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 13 achieved by using wrapper functions on top of the C implementation. The author of the 14profile library then assumes that it is reasonable only to provide profile functions for the C 15binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 16to be small. However, if the wrapper functions are not in the profiling library, then none 17of the profiled entry points will be undefined when the profiling library is called. Therefore 18 none of the profiling code will be included in the image. When the standard MPI library 19is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 20the MPI functions. The overall effect is that the code will link successfully, but will not be 21profiled. 22

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 copied out of the base library and into the profiling one using a tool such as **ar**.

27

# <sup>28</sup> 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 17.1.5 on page 609.

 $\frac{34}{35}$ 

# 14.2.8 Multiple Levels of Interception

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

40 41

42

43

- 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.

<sup>47</sup> Note, however, that it is possible to use the scheme above to implement a multi-level <sup>48</sup> 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

MPI implementations often use internal variables to control their operation and performance. Understanding and manipulating these variables can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation. The interface is split into two parts: the first part provides information about and supports the setting of control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information 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 communication 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\_.

On success, all MPI tool information interface routines return MPI\_SUCCESS, otherwise 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. 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

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

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 10 (end users, performance tuners or MPI implementors) and a relative measure of level of 11 detail (basic, detailed or all). These verbosity levels are described by a single integer. 12Table 14.1 lists the constants for all possible verbosity levels. The values of the con-13 stants are monotonic in the order listed in the table; i.e., MPI\_T\_VERBOSITY\_USER\_BASIC 14< MPI\_T\_VERBOSITY\_USER\_DETAIL < ... < MPI\_T\_VERBOSITY\_MPIDEV\_ALL. 15

16		
17	MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users
18	MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users
18	MPI_T_VERBOSITY_USER_ALL	All remaining information of interest to users
20	MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning
20 21	MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
21	MPI_T_VERBOSITY_TUNER_ALL	All remaining information required for tuning
22	MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
	MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
24 25	MPI_T_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors

Table 14.1: MPI tool information interface verbosity levels

# 28 2930

 $^{31}$ 

### 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 32 or performance property of the MPI implementation. A variable may refer to a specific 33 MPI object such as a communicator, datatype, or one-sided communication window, or the 34 variable may refer more generally to the MPI environment of the process. Except for the 35 last case, the variable must be bound to exactly one MPI object before it can be used. 36 Table 14.2 lists all MPI object types to which an MPI tool information interface variable 37 can be bound, together with the matching constant that MPI tool information interface 38 routines return to identify the object type. 39

40

41

42

43

44

45

46

47

48

Rationale. Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations that use 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 bound, 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

3 4

5

6

7

8

9

1

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object

Table 14.2: Constants to identify associations of variables

then be applied to as many MPI objects of the respective type as created during the program's execution. (End of rationale.)

# 14.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an IN/OUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most n-1 of the string's characters into the buffer, followed by a null terminator. If the returned string's length is greater than or equal to n, the string will be truncated to n-1 characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length 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. 

### Initialization and Finalization 14.3.4

The MPI tool information interface requires a separate set of initialization and finalization routines.

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)

 $^{24}$ 

1 All programs or tools that use the MPI tool information interface must initialize the  $\mathbf{2}$ MPI tool information interface in the processes that will use the interface before calling 3 any other of its routines. A user can initialize the MPI tool information interface by calling 4 MPI\_T\_INIT\_THREAD, which can be called multiple times. In addition, this routine initial- $\mathbf{5}$ izes the thread environment for all routines in the MPI tool information interface. Calling 6 this routine when the MPI tool information interface is already initialized has no effect  $\overline{7}$ beyond increasing the reference count of how often the interface has been initialized. The 8 argument required is used to specify the desired level of thread support. The possible values 9 and their semantics are identical to the ones that can be used with MPI\_INIT\_THREAD 10 listed in Section 12.4. The call returns in provided information about the actual level of 11thread support that will be provided by the MPI implementation for calls to MPI tool 12information 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 MPI\_INIT. If the MPI tool information interface is used before MPI\_INIT has been called, the user is responsible for ensuring that the MPI tool information interface is initialized on all processes it is used in. Processes created by the MPI implementation during MPI\_INIT inherit the status of the MPI tool information interface (whether it is initialized or not as well as all active sessions and handles) from the process from which they are created.

Processes created at runtime as a result of calls to MPI's dynamic process management
 require their own initialization before they can use the MPI tool information interface.

Advice to users. If MPI\_T\_INIT\_THREAD is called before MPI\_INIT\_THREAD, 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.)

31 32 33

21

22

23

24

25 26

27

28

29

30

34

35 36

37

45

MPI\_T\_FINALIZE( )

# int MPI\_T\_finalize(void)

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

<sup>46</sup> MPI\_T\_INIT\_THREAD up to the current point of execution, the MPI tool information in-<sup>47</sup> terface is no longer initialized. The interface can be reinitialized by subsequent calls to <sup>48</sup> 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.

# 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 separate 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
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_LONG_LONG
MPI_COUNT
MPI_CHAR
MPI_DOUBLE

Table 14.3: MPI datatypes that can be used by the MPI tool information interface

*Rationale.* 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).

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. (*End of rationale.*)

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 to 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 48

 $\overline{7}$ 

	that can be queried using M	
MPI_T_EN	IUM_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 $name\xspace$ (integer)
int MPI_7	[_enum_get_info(MPI_T_en *name_len)	num enumtype, int *num, char *name, int
The r	—	a name of at least length one. This name must be
using MPI	s associated with individual _T_ENUM_GET_ITEM.	for enumerations that the MPI implementation uses values in each enumeration enumtype can be queried
using MPI	s associated with individual _T_ENUM_GET_ITEM.	values in each enumeration enumtype can be queried index, value, name, name_len)
using MPI	s associated with individual _T_ENUM_GET_ITEM.	values in each enumeration enumtype can be queried
using MPI MPI_T_EN	s associated with individual _T_ENUM_GET_ITEM. NUM_GET_ITEM(enumtype,	<pre>values in each enumeration enumtype can be queried index, value, name, name_len) enumeration to be queried (handle)</pre>
using MPI MPI_T_EN IN	s associated with individual _T_ENUM_GET_ITEM. NUM_GET_ITEM(enumtype, enumtype	<pre>values in each enumeration enumtype can be queried index, value, name, name_len) enumeration to be queried (handle) number of the value to be queried in this enumeration</pre>
using MPI MPI_T_EN IN IN	s associated with individual _T_ENUM_GET_ITEM. NUM_GET_ITEM(enumtype, enumtype index	<pre>values in each enumeration enumtype can be queried index, value, name, name_len) enumeration to be queried (handle) number of the value to be queried in this enumeration (integer)</pre>
using MPI MPI_T_EN IN IN OUT	s associated with individual _T_ENUM_GET_ITEM. NUM_GET_ITEM(enumtype, enumtype index value	<pre>values in each enumeration enumtype can be queried index, value, name, name_len) enumeration to be queried (handle) number of the value to be queried in this enumeration (integer) variable value (integer) buffer to return the string containing the name of the</pre>
USING MPI MPI_T_EN IN IN OUT OUT INOUT	s associated with individual _T_ENUM_GET_ITEM. NUM_GET_ITEM(enumtype, enumtype index value name name_len	<pre>values in each enumeration enumtype can be queried index, value, name, name_len) enumeration to be queried (handle) number of the value to be queried in this enumeration (integer) variable value (integer) buffer to return the string containing the name of the enumeration item (string) length of the string and/or buffer for name (integer) num enumtype, int index, int *value, char</pre>

 $\mathbf{2}$  $\mathbf{5}$  $\overline{7}$ 

 $^{21}$ 

 $^{31}$ 

# 14.3.6 Control Variables

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an "eager limit," i.e., an upper bound on the size of messages sent or received using an eager protocol.

# Control Variable Query Functions

An MPI implementation exports a set of N control variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any control variables, otherwise the provided control 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 control 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 control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (*End of advice to users.*)

The following function can be used to query the number of control variables, *num\_cvar*:

# MPI\_T\_CVAR\_GET\_NUM(num\_cvar) OUT num\_cvar returns number of control variables (integer) int MPI\_T\_cvar\_get\_num(int \*num\_cvar) The function MPI\_T\_CVAR\_GET\_INFO provides access to additional information for each variable.

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

12	MPI_T_C\	/AR_GET_INFO(cvar_index, n desc_len, bind, scope)	ame, name_len, verbosity, datatype, enumtype, desc,	
3 4 5	IN	cvar_index	index of the control variable to be queried, value be- tween 0 and $num\_cvar - 1$ (integer)	
6 7	OUT	name	buffer to return the string containing the name of the control variable (string)	
8	INOUT	name_len	length of the string and/or buffer for name (integer) $% \left( {{\left[ {{{\left[ {{{\left[ {{\left[ {{\left[ {{\left[ {{{\left[ {{{\left[ {{{\left[ {{\left[ {{{\left[ {{{\left[ {{{\left[ {{{\left[ {{{}}}} \right]}}}} \right.$	
9 10	OUT	verbosity	verbosity level of this variable (integer)	
11 12	OUT	datatype	MPI data type of the information stored in the control variable (handle)	
13 14	OUT	enumtype	optional descriptor for enumeration information (han- dle)	
15 16	OUT	desc	buffer to return the string containing a description of the control variable (string)	
17 18	INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)	
19 20	OUT	bind	type of MPI object to which this variable must be bound (integer)	
21 22 23	OUT	scope	scope of when changes to this variable are possible (integer)	
24 25 26 27	int MPI_T	*verbosity, MPI_Data	index, char *name, int *name_len, int atype *datatype, MPI_T_enum *enumtype, char a, int *bind, int *scope)	
28 29 30 31 32	calls to thi informatio The a	is routine that query informat n. An MPI implementation is	AR_GET_INFO for a particular variable, subsequent tion about the same variable must return the same a not allowed to alter any of the returned values. are used to return the name of the control variable	
33 34 25	If completed successfully, the routine is required to return a name of at least length one. The name must be unique with respect to all other names for control variables used			
35 36	by the MPI implementation. The argument verbosity returns the verbosity level of the variable (see Section 14.3.1)			
37	The argument verbosity returns the verbosity level of the variable (see Section 14.3.1). The argument datatype returns the MPI datatype that is used to represent the control			
38	variable.			
39	If the variable is of type $MPI\_INT,$ $MPI$ can optionally specify an enumeration for the			
40 41	-	e e	return it in enumtype. In this case, MPI returns an	
42			be used to gather more information as described in set to MPI_T_ENUM_NULL. If the datatype is not	
43		, .	he null pointer, no enumeration type is returned.	
44			e used to return a description of the control variable	
45		ed in Section 14.3.3.		
46			l. If an MPI implementation does not to return a	
47 48	-	n, the first character for desc none at the return of this call.	must be set to the null character and desc_len must	

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).

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 either to 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_CONSTANT	read-only, value is constant
MPI_T_SCOPE_READONLY	read-only, cannot be written, but can change
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

# Example 14.4

The following example shows how the MPI tool information interface can be used to query and to print the names of all available control variables.

```
#include <stdio.h>
#include <stdlib.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;
```

 $^{31}$ 

```
1
\mathbf{2}
        err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
3
        if (err!=MPI_SUCCESS)
4
          return err;
5
6
        err=MPI_T_cvar_get_num(&num);
\overline{7}
        if (err!=MPI_SUCCESS)
8
          return err;
9
10
        for (i=0; i<num; i++) {</pre>
11
          namelen=100;
12
          err=MPI_T_cvar_get_info(i, name, &namelen,
13
                    &verbose, &datatype, NULL,
14
                    NULL, NULL, /*no description */
15
                    &bind, &scope);
16
          if (err!=MPI_SUCCESS) return err;
17
          printf("Var %i: %s\n", i, name);
18
        }
19
20
        err=MPI_T_finalize();
21
        if (err!=MPI_SUCCESS)
22
          return 1;
23
        else
^{24}
          return 0;
25
      }
26
27
      Handle Allocation and Deallocation
28
      Before reading or writing the value of a variable, a user must first allocate a handle of type
29
      MPI_T_cvar_handle for the variable by binding it to an MPI object (see also Section 14.3.2).
30
^{31}
            Rationale.
                         Handles used in the MPI tool information interface are distinct from
32
           handles used in the remaining parts of the MPI standard because they must be usable
33
           before MPI_INIT and after MPI_FINALIZE. Further, accessing handles, in particular
34
           for performance variables, can be time critical and having a separate handle space
35
           enables optimizations. (End of rationale.)
36
37
38
      MPI_T_CVAR_HANDLE_ALLOC(cvar_index, obj_handle, handle, count)
39
        IN
                  cvar_index
                                                index of control variable for which handle is to be al-
40
                                                located (index)
41
42
        IN
                  obj_handle
                                                reference to a handle of the MPI object to which this
43
                                                variable is supposed to be bound (pointer)
44
        OUT
                  handle
                                                allocated handle (handle)
45
        OUT
                                                number of elements used to represent this variable (in-
                  count
46
                                                teger)
47
48
```

# 

This routine binds the control variable specified by the argument index to an MPI object. The object is passed in the argument obj\_handle as an address to a local variable that stores the object's handle. The argument obj\_handle is ignored if the MPI\_T\_CVAR\_GET\_INFO call for this control variable returned MPI\_T\_BIND\_NO\_OBJECT in the argument bind. The handle allocated to reference the variable is returned in the argument handle. Upon successful return, count contains the number of elements (of the datatype returned by a previous MPI\_T\_CVAR\_GET\_INFO call) used to represent this variable.

Advice to users. The count can be different based on the MPI object to which the control variable was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.

It is not portable to pass references to predefined MPI object handles, such as MPI\_COMM\_WORLD to this routine, since their implementation depends on the MPI library. Instead, such object handles should be stored in a local variable and the address of this local variable should be passed into MPI\_T\_CVAR\_HANDLE\_ALLOC. (*End of advice to users.*)

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 that the bind argument returned by MPI\_T\_CVAR\_GET\_INFO equals MPI\_T\_BIND\_NO\_OBJECT, the argument obj\_handle is ignored.

MPI_T_CV	AR_HANDLE_FREE(handle)	
INOUT	handle	handle to be freed (handle)

# int MPI\_T\_cvar\_handle\_free(MPI\_T\_cvar\_handle \*handle)

When a handle is no longer needed, a user of the MPI tool information interface should call MPI\_T\_CVAR\_HANDLE\_FREE to free the handle and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to MPI\_T\_CVAR\_HANDLE\_NULL.

Control Variable Access Functions

MPI\_T\_CVAR\_READ(handle, buf)

IN	handle	handle to the control variable to be read (handle)	43
OUT	buf	initial address of storage location for variable value	$44 \\ 45$
		(choice)	46

int MPI\_T\_cvar\_read(MPI\_T\_cvar\_handle handle, void\* buf)

 $\mathbf{2}$ 

1 This routine queries the value of the control variable identified by the argument handle  $\mathbf{2}$ and stores the result in the buffer identified by the parameter buf. The user must ensure that 3 the buffer is of the appropriate size to hold the entire value of the control variable (based on 4 the returned datatype and count from prior corresponding calls to MPI\_T\_CVAR\_GET\_INFO  $\mathbf{5}$ and MPI\_T\_CVAR\_HANDLE\_ALLOC, respectively). 6 7MPI\_T\_CVAR\_WRITE(handle, buf) 8 9 IN handle handle to the control variable to be written (handle) 10 IN buf initial address of storage location for variable value 11 (choice) 1213int MPI\_T\_cvar\_write(MPI\_T\_cvar\_handle handle, const void\* buf) 1415This routine sets the value of the control variable identified by the argument handle to 16the data stored in the buffer identified by the parameter buf. The user must ensure that the 17buffer is of the appropriate size to hold the entire value of the control variable (based on the 18returned datatype and count from prior corresponding calls to MPI\_T\_CVAR\_GET\_INFO 19and MPI\_T\_CVAR\_HANDLE\_ALLOC, respectively). 20If the variable has a global scope (as returned by a prior corresponding 21MPI\_T\_CVAR\_GET\_INFO call), any write call to this variable must be issued by the user 22in all connected (as defined in Section 10.5.4) MPI processes. If the variable has group 23scope, any write call to this variable must be issued by the user in all MPI processes in  $^{24}$ the group, which must be described by the MPI implementation in the description by the 25MPI\_T\_CVAR\_GET\_INFO. 26In both cases, the user must ensure that the writes in all processes are consistent. If 27the scope is either MPI\_T\_SCOPE\_ALL\_EQ or MPI\_T\_SCOPE\_GROUP\_EQ this means that the 28 variable in all processes must be set to the same value. 29If it is not possible to change the variable at the time the call is made, the function 30returns either MPI\_T\_ERR\_CVAR\_SET\_NOT\_NOW, if there may be a later time at which the  $^{31}$ variable could be set, or MPI\_T\_ERR\_CVAR\_SET\_NEVER, if the variable cannot be set for the 32 remainder of the application's execution. 33 34Example: Reading the Value of a Control Variable 35 36 Example 14.5 37 The following example shows a routine that can be used to query the value with a 38 control variable with a given index. The example assumes that the variable is intended to 39 be bound to an MPI communicator. 40 41 int getValue\_int\_comm(int index, MPI\_Comm comm, int \*val) { 42int err, count; 43 MPI\_T\_cvar\_handle handle; 44 45/\* This example assumes that the variable index \*/ 46 /\* can be bound to a communicator \*/ 4748

```
err=MPI_T_cvar_handle_alloc(index,&comm,&handle,&count);
if (err!=MPI_SUCCESS) return err;
/* The following assumes that the variable is */
/* represented by a single integer */
err=MPI_T_cvar_read(handle,val);
if (err!=MPI_SUCCESS) return err;
err=MPI_T_cvar_handle_free(&handle);
return err;
}
```

### Performance Variables 14.3.7

The following section focuses on the ability to list and to 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 that is assigned to the variable the first time that it is used 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 protect against this overflow, e.g., by frequently reading and resetting the variable value. (End of advice to users.)

MPI implementations should use large enough datatypes Advice to implementors. 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 43 44this 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 that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

1

2

3

4

5 6

7

9 10

11

1213 14

15

16

17

18

19

2021

22

23

24

2526

27

2829

30

31

32

33

34

3536

37

38

39 40

41 42

4546

47

### 1 MPI\_T\_PVAR\_CLASS\_LEVEL 2 A performance variable in this class represents a value that describes the utilization 3 level of a resource. The value of a variable of this class can change at any time to match 4 the current utilization level of the resource. Values returned from variables in this class 5are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, 6 MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value 7 is the current utilization level of the resource at the time that the starting value is 8 set. MPI implementations must ensure that variables of this class cannot overflow. 9 MPI\_T\_PVAR\_CLASS\_SIZE 10 A performance variable in this class represents a value that is the fixed size of a 11 resource. Values returned from variables in this class are non-negative and rep-12resented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, 13 MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utiliza-14tion level of the resource at the time that the starting value is set. MPI implementa-15tions must ensure that variables of this class cannot overflow. 1617 MPI\_T\_PVAR\_CLASS\_PERCENTAGE 18 The value of a performance variable in this class represents the percentage utiliza-19 tion of a finite resource. The value of a variable of this class can change at any 20time to match the current utilization level of the resource. It will be returned as an 21MPI\_DOUBLE datatype. The value must always be between 0.0 (resource not used at 22 all) and 1.0 (resource completely used). The starting value is the current percent-23age utilization level of the resource at the time that the starting value is set. MPI 24implementations must ensure that variables of this class cannot overflow. 25 MPI\_T\_PVAR\_CLASS\_HIGHWATERMARK 26A performance variable in this class represents a value that describes the high water-27mark utilization of a resource. The value of a variable of this class is non-negative 28and grows monotonically from the initialization or reset of the variable. It can be rep-29 resented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, 30 MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utiliza-31tion level of the resource at the time that the starting value is set. MPI implementa-32 tions must ensure that variables of this class cannot overflow. 33

34 35

36

37

38

39

40

41

42 43

# • MPI\_T\_PVAR\_CLASS\_LOWWATERMARK

A performance variable in this class represents a value that describes the low watermark utilization of a resource. The value of a variable of this class is non-negative and decreases 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\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI\_T\_PVAR\_CLASS\_COUNTER

<sup>44</sup> A performance variable in this class counts the number of occurrences of a specific <sup>45</sup> event (e.g., the number of memory allocations within an MPI library). The value of <sup>46</sup> a variable of this class increases monotonically from the initialization or reset of the <sup>47</sup> performance variable by one for each specific event that is observed. Values must <sup>48</sup> be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG. The starting value for variables of this class is 0. Variables of this class can overflow.

# • MPI\_T\_PVAR\_CLASS\_AGGREGATE

The value of a performance variable in this class is an an aggregated value that represents a sum of arguments processed during a specific event (e.g., the amount of memory allocated by all memory allocations). This class is similar to the counter class, but instead of counting individual events, the value can be incremented by arbitrary amounts. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG, MPI\_DOUBLE. The starting value for variables of this class is 0. Variables of this class can overflow.

# • MPI\_T\_PVAR\_CLASS\_TIMER

The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section of the MPI library. This class has the same basic semantics as MPI\_T\_PVAR\_CLASS\_AGGREGATE, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, 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 that represent 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.

# 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 to delete a variable once it has been added to the set. When a variable becomes 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:

 $\overline{7}$ 

 $^{24}$ 

1	MPI_T_PVAR_GET_NUM(num_pvar)						
2	OUT	num_pvar	returns number of performance variables (integer)				
$\frac{3}{4}$							
5	int MPI_7	<pre>int MPI_T_pvar_get_num(int *num_pvar)</pre>					
6	The f	The function MPI_T_PVAR_GET_INFO provides access to additional information for					
7 8	each varia	each variable.					
9							
10 11	MPI_T_P	MPI_T_PVAR_GET_INFO(pvar_index, name, name_len, verbosity, varclass, datatype, enumtype, desc, desc_len, bind, readonly, continuous, atomic)					
12 13	IN	pvar_index	index of the performance variable to be queried be- tween 0 and $num_pvar - 1$ (integer)				
14 15 16	OUT	name	buffer to return the string containing the name of the performance variable (string)				
17	INOUT	name_len	length of the string and/or buffer for $name$ (integer)				
18	OUT	verbosity	verbosity level of this variable (integer)				
19	OUT	var_class	class of performance variable (integer)				
20 21 22	OUT	datatype	MPI data type of the information stored in the performance variable (handle)				
23 24	OUT	enumtype	optional descriptor for enumeration information (han- dle)				
25 26	OUT	desc	buffer to return the string containing a description of the performance variable (string)				
27	INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)				
28 29 30	OUT	bind	type of MPI object to which this variable must be bound (integer)				
31 32	OUT	readonly	flag indicating whether the variable can be written/reset (integer)				
33 34	OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)				
35 36 37	OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)				
38 39 40	int MPI_7	<pre>int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>					
41 42			, char *desc, int *desc_len, int *bind, continuous, int *atomic)				
43 44 45 46 47 48	After a successful call to MPI_T_PVAR_GET_INFO for a particular variable, subsequent calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values. The arguments name and name_len are used to return the name of the performance variable as described in Section 14.3.3. If completed successfully, the routine is required to return a name of at least length one.						

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple 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 to gather more information as described in Section 14.3.5. Otherwise, enumtype is set to MPI\_T\_ENUM\_NULL. If the datatype is not MPI\_INT or the argument enumtype is the null pointer, no emumeration type is returned.

Returning a description is optional. If an MPI implementation does 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 by the user.

Upon return, the argument **atomic** is set to zero if the variable cannot be read and reset atomically. 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 that access 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.

# MPI\_T\_PVAR\_SESSION\_CREATE(session) OUT session identifier of performance session (handle)

# int MPI\_T\_pvar\_session\_create(MPI\_T\_pvar\_session \*session)

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.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

 $\frac{44}{45}$ 

1	MPI_T_PVAR_SESSION_FREE(session)					
2 3	INOUT	session	identifier of performance experiment session (handle)			
4 5	int MPI_T	<pre>int MPI_T_pvar_session_free(MPI_T_pvar_session *session)</pre>				
6 7 8 9	This call frees an existing session. Calls to the MPI tool information interface can no longer be made within the context of a session after it is freed. On a successful return, MPI sets the session identifier to MPI_T_PVAR_SESSION_NULL.					
10 11	Handle Allocation and Deallocation					
12 13 14	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).					
15 16	MPI_T_PV	AR_HANDLE_ALLOC(session,	pvar_index, obj_handle, handle, count)			
17	IN	session	identifier of performance experiment session (handle)			
18 19	IN	pvar_index	index of performance variable for which handle is to be allocated (integer)			
20 21 22	IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)			
23	OUT	handle	allocated handle (handle)			
24 25 26	OUT	count	number of elements used to represent this variable (in- teger)			
27 28 29	int MPI_T	<pre>int MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,</pre>				
30 31 32 33 34 35 36 37	This routine binds the performance variable specified by the argument index to an MPI object in the session identified by the parameter session. The object is passed in the argument obj_handle as an address to a local variable that stores the object's handle. The argument obj_handle is ignored if the MPI_T_PVAR_GET_INFO call for this performance variable returned MPI_T_BIND_NO_OBJECT in the argument bind. The handle allocated to reference the variable is returned in the argument handle. Upon successful return, count contains the number of elements (of the datatype returned by a previous MPI_T_PVAR_GET_INFO call) used to represent this variable.					
38 39 40 41	Advice to users. The count can be different based on the MPI object to which the performance variable was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.					
42 43 44 45 46 47 48	It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI_T_PVAR_HANDLE_ALLOC. ( <i>End of advice to users.</i> )					

The v	value of index shou	ld be in the range 0 to $num_pvar - 1$ , where $num_pvar$ is the	1			
	-	ance variables as determined from a prior call to	2			
MPI_T_P	/AR_GET_NUM. T	The type of the MPI object it references must be consistent	3			
with the t	ype returned in the	e bind argument in a prior call to MPI_T_PVAR_GET_INFO.	4			
In the	e case the bind $\operatorname{arg}$	ument equals $MPI_T_BIND_NO_OBJECT$ , the argument	5			
obj_handle	is ignored.		6			
			7			
			8			
MPI_I_PV	/AR_HANDLE_FRE	Et(session, handle)	9			
IN	session	identifier of performance experiment session (handle)	10			
INOUT	handle	handle to be freed (handle)	11			
into o r	nunare		12			
	r	A (MDT T more consist and MDT T more bandle	13			
int MPI_I	-	ee(MPI_T_pvar_session session, MPI_T_pvar_handle	14			
	<pre>*handle)</pre>		15			
When	a handle is no long	ger needed, a user of the MPI tool information interface should	16			
call MPI_	T_PVAR_HANDLE_	FREE to free the handle in the session identified by the pa-	17			
rameter se	ession and the asso	ciated resources in the MPI implementation. On a successful	18			
return, MI	<b>PI</b> sets the handle t	O MPI_T_PVAR_HANDLE_NULL.	19			
			20			
Starting ar	nd Stopping of Perfo	ormance Variables	21			
•			22			
		have the continuous flag set during the query operation are	23			
		a handle has been allocated. Such variables may be queried at	24			
		e started or stopped by the user. All other variables are in a	25			
		dle has been allocated; their values are not updated until they	26			
have been	started by the user	r.	27			
			28			
	/AR_START(sessior	handle)	29			
	,	,	30			
IN	session	identifier of performance experiment session (handle)	31			
IN	handle	handle of a performance variable (handle)	32			
		-	33			
int MPT 1	pvar start(MPT	_T_pvar_session session, MPI_T_pvar_handle handle)	34			
	-		35			
		e performance variable with the handle identified by the pa-	36			
		identified by the parameter session.	37			
		AR_ALL_HANDLES is passed in handle, the MPI implementation	38			
attempts to start all variables within the session identified by the parameter session for						
which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all						
variables are started successfully, otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned.						
	Continuous variables and variables that are already started are ignored when					
MPI_T_PVAR_ALL_HANDLES is specified.						
			44			
			45			
			46			
			47			
			48			

	PVAR_STOP(session	,
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)
ter han If th attempt which h variables Continu MPI_T_F	dle in the session ide ne constant MPI_T_PV s to stop all variabl andles have been all s are stopped success	
MPI_T_	PVAR_READ(session	, handle, buf)
IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)
OUT	buf	initial address of storage location for variable value (choice)
int MPI	_T_pvar_read(MPI_ void* buf)	T_pvar_session session, MPI_T_pvar_handle handle,
nandle F ouffer id s of the he data MPI_T_ The	handle in the session is lentified by the para e appropriate size to atype and count retu PVAR_GET_INFO and	D call queries the value of the performance variable with the identified by the parameter <b>session</b> and stores the result in the ameter <b>buf</b> . The user is responsible to ensure that the buffer hold the entire value of the performance variable (based on urned by the corresponding previous calls to ad MPI_T_PVAR_HANDLE_ALLOC, respectively). AR_ALL_HANDLES cannot be used as an argument for the func-
мрі т	PVAR_WRITE(sessio	n handle, buf)
IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)
IN	buf	initial address of storage location for variable value (choice)
int MPI	_T_pvar_write(MPI const void*	_T_pvar_session session, MPI_T_pvar_handle handle, * buf)

The MPI\_T\_PVAR\_WRITE call attempts to write the value of the performance variable with the handle identified by the parameter handle in the session identified by the parameter session. The value to be written is passed in the buffer identified by the parameter buf. The user must ensure that the buffer is of the appropriate size to hold the entire value of the performance variable (based on the datatype and count returned by the corresponding previous calls to MPI\_T\_PVAR\_GET\_INFO and MPI\_T\_PVAR\_HANDLE\_ALLOC, respectively).

If it is not possible to change the variable, the function returns MPI\_T\_ERR\_PVAR\_NO\_WRITE.

The constant MPI\_T\_PVAR\_ALL\_HANDLES cannot be used as an argument for the function MPI\_T\_PVAR\_WRITE.

MPI_T_PVAR_RESET(session, handle)		
IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)

int MPI\_T\_pvar\_reset(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle)

The MPI\_T\_PVAR\_RESET call sets the performance variable with the handle identified by the parameter handle to its starting value specified in Section 14.3.7. If it is not possible to change the variable, the function returns MPI\_T\_ERR\_PVAR\_NO\_WRITE.

If the constant MPI\_T\_PVAR\_ALL\_HANDLES is passed in handle, the MPI implementation attempts to reset all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI\_SUCCESS if all variables are reset successfully, otherwise MPI\_T\_ERR\_PVAR\_NO\_WRITE is returned. Readonly variables are ignored when MPI\_T\_PVAR\_ALL\_HANDLES is specified.

MPI\_T\_PVAR\_READRESET(session, handle, buf)

IN	session	identifier of performance experiment session (handle)	30
IN	handle	handle of a performance variable (handle)	31 32
OUT	buf	initial address of storage location for variable value	33
		(choice)	34

#### int MPI\_T\_pvar\_readreset(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle, void\* buf)

This call atomically combines the functionality of MPI\_T\_PVAR\_READ and MPI\_T\_PVAR\_RESET with the same semantics as if these two calls were called separately. If atomic operations on this variable are not supported, this routine returns MPI\_T\_ERR\_PVAR\_NO\_ATOMIC.

The constant MPI\_T\_PVAR\_ALL\_HANDLES cannot be used as an argument for the function MPI\_T\_PVAR\_READRESET.

Advice to implementors. Sampling-based tools rely on the ability to call the MPI tool information interface, in particular routines to start, stop, read, write and reset performance variables, from any program context, including asynchronous contexts such

1  $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

18 19

20

21

22

23

 $^{24}$ 

25

26

2728

35 36

37

38

39

40

41

42

43

4445

46

47

as signal handlers. MPI implementations should strive, if possible in their particular environment, to enable these usage scenarios for all or a subset of the routines mentioned above. If implementing only a subset, the read, write, and reset routines are typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI\_T\_PVAR\_GET\_INFO. (*End of advice to implementors.*)

Rationale. All routines to read, to write or to reset performance variables require the session argument. This requirement keeps the interface consistent and allows the use of 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

## <sup>18</sup> Example 14.6

<sup>19</sup> The following example shows a sample tool to identify receive operations that occur <sup>20</sup> during times with long message queues. This examples assumes that the MPI implementa-<sup>21</sup> tion exports a variable with the name MPI\_T\_UMQ\_LENGTH to represent the current length <sup>22</sup> of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI <sup>23</sup> 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.

```
29
30
```

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.

```
34
     #include <stdio.h>
35
     #include <stdlib.h>
36
     #include <string.h>
37
     #include <assert.h>
38
     #include <mpi.h>
39
40
     /* Global variables for the tool */
^{41}
     static MPI_T_pvar_session session;
42
     static MPI_T_pvar_handle handle;
43
44
     int MPI_Init(int *argc, char ***argv ) {
45
     int err, num, i, index, namelen, verbosity;
46
              int var_class, bind, threadsup;
47
     int readonly, continuous, atomic, count;
48
     char name [18];
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

14 15 16

```
1
MPI_Comm comm;
                                                                                    2
MPI_Datatype datatype;
MPI_T_enum enumtype;
                                                                                    4
err=PMPI_Init(argc,argv);
                                                                                    5
                                                                                    6
if (err!=MPI_SUCCESS) return err;
                                                                                    7
                                                                                    8
err=PMPI_T_init_thread(MPI_THREAD_SINGLE,&threadsup);
                                                                                    9
if (err!=MPI_SUCCESS) return err;
                                                                                    10
                                                                                    11
err=PMPI_T_pvar_get_num(&num);
if (err!=MPI_SUCCESS) return err;
                                                                                    12
                                                                                    13
index=-1;
                                                                                    14
i=0:
                                                                                    15
while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
                                                                                    16
/* Pass a buffer that is at least one character longer than */
                                                                                    17
/* the name of the variable being searched for to avoid */
                                                                                    18
/* finding variables that have a name that has a prefix */
                                                                                    19
/* equal to the name of the variable being searched. */
namelen=18;
                                                                                    20
                                                                                    21
err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
                                                                                    22
&var_class, &datatype, &enumtype, NULL, NULL, &bind,
&readonly, &continuous, &atomic);
                                                                                    23
                                                                                    24
if (strcmp(name, "MPI_T_UMQ_LENGTH")==0) index=i;
                                                                                    25
i++; }
                                                                                    26
if (err!=MPI_SUCCESS) return err;
                                                                                    27
/* this could be handled in a more flexible way for a generic tool */
                                                                                    28
                                                                                    29
assert(index>=0);
assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
                                                                                    30
                                                                                    31
assert(datatype==MPI_INT);
                                                                                    32
assert(bind==MPI_T_BIND_MPI_COMM);
                                                                                    33
                                                                                    34
/* Create a session */
err=PMPI_T_pvar_session_create(&session);
                                                                                    35
if (err!=MPI_SUCCESS) return err;
                                                                                    36
                                                                                    37
/* Get a handle and bind to MPI_COMM_WORLD */
                                                                                    38
                                                                                    39
comm=MPI_COMM_WORLD;
err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
                                                                                    40
                                                                                    41
if (err!=MPI_SUCCESS) return err;
                                                                                    42
/* this could be handled in a more flexible way for a generic tool */
                                                                                    43
                                                                                    44
assert(count==1);
                                                                                    45
/* Start variable */
                                                                                    46
                                                                                    47
err=PMPI_T_pvar_start(session, handle);
                                                                                    48
if (err!=MPI_SUCCESS) return err;
```

```
1
\mathbf{2}
     return MPI_SUCCESS;
3
     }
4
\mathbf{5}
     Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the
6
     tool reads the unexpected queue length through the matching performance variable and
7
     compares it against a predefined threshold.
8
9
     #define THRESHOLD 5
10
^{11}
     int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
12
                                  MPI_Comm comm, MPI_Status *status)
13
     {
14
     int value, err;
15
16
     if (comm==MPI_COMM_WORLD) {
17
     err=PMPI_T_pvar_read(session, handle, &value);
^{18}
     if ((err==MPI_SUCCESS) && (value>THRESHOLD))
19
     {
20
                                 /* tool identified receive called with long UMQ */
21
     /* execute tool functionality, */
22
     /* e.g., gather and print call stack */
23
     }
^{24}
     }
25
26
     return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
27
     }
28
29
     Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information inter-
30
     face is finalized.
31
32
     int MPI_Finalize()
33
     {
34
     int err;
35
     err=PMPI_T_pvar_handle_free(session, &handle);
36
     err=PMPI_T_pvar_session_free(&session);
37
     err=PMPI_T_finalize();
38
     return PMPI_Finalize();
39
     }
40
41
     14.3.8 Variable Categorization
42
     MPI implementations can optionally group performance and control variables into categories
43
     to express logical relationships between various variables. For example, an MPI implemen-
44
     tation could group all control and performance variables that refer to message transfers in
45
     the MPI implementation and thereby distinguish them from variables that refer to local
46
```

47 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 to 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.

Similarly, MPI implementations are allowed to add variables to categories, but they are not allowed to remove variables from categories or change the order in which they are returned.

The following function can be used to query the number of control variables, N.

#### MPI\_T\_CATEGORY\_GET\_NUM(num\_cat)

			• • • •
OUT	num_cat	current number of categories (	integer)
001	nunn_cut	current number of categories (	mucger

#### int MPI\_T\_category\_get\_num(int \*num\_cat)

Individual category information can then be queried by calling the following function:

 $\overline{7}$ 

1 2	MPI_T_CA	TEGORY_GET_INFO(cat_inde num_pvars, num_categori	ex, name, name_len, desc, desc_len, num_cvars, es)
3	IN	cat_index	index of the category to be queried (integer)
4 5 6	OUT	name	buffer to return the string containing the name of the category (string)
7	INOUT	name_len	length of the string and/or buffer for name (integer)
8 9 10	OUT	desc	buffer to return the string containing the description of the category (string)
11	INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)
12	OUT	num_cvars	number of control variables in the category (integer)
13 14 15	OUT	num_pvars	number of performance variables in the category (in-teger)
16 17 18	OUT	num_categories	number of categories contained in the category (integer)
19 20 21 22 23 24 25 26 27 28 29 30 31	The a described i The re unique with The au described i Return description be set to o	char *desc, int *desc int *num_categories) rguments name and name_len n Section 14.3.3. butine is required to return a h respect to all other names for rguments desc and desc_len ar n Section 14.3.3. hing a description is optional. t, the first character for desc m ne at the return of this call.	at_index, char *name, int *name_len, c_len, int *num_cvars, int *num_pvars, a are used to return the name of the category as name of at least length one. This name must be or categories used by the MPI implementation. e used to return the description of the category as If an MPI implementation decides not to return a nust be set to the null character and desc_len must
32 33 34 35	categories		control variables, performance variables and other gory in the arguments num_cvars, num_pvars, and
36	MPI_T_CA	TEGORY_GET_CVARS(cat_in	idex, len, indices)
37 38 39	IN	cat_index	index of the category to be queried, in the range $[0,N-1]$ (integer)
40	IN	len	the length of the indices array (integer)
41 42 43	Ουτ	indices	an integer array of size len, indicating control variable indices (array of integers)
44 45 46			<pre>cat_index, int len, int indices[]) can be used to query which control variables are</pre>
40 47 48			tegory contains zero or more control variables.

MPI_T_CA	TEGORY_GET_PVARS(cat_in	dex,len,indices)	1
IN	cat_index	index of the category to be queried, in the range $[0,N-1]$ (integer)	2 3 4
IN	len	the length of the indices array (integer)	4 5
OUT	indices	an integer array of size len, indicating performance variable indices (array of integers)	6 7 8
int MPI_T_	_category_get_pvars(int c	at_index, int len, int indices[])	9 10
		an be used to query which performance variables A category contains zero or more performance	10 11 12 13 14
MPI_T_CA	TEGORY_GET_CATEGORIES	(cat_index,len,indices)	15 16
IN	cat_index	index of the category to be queried, in the range $[0, N-1]$ (integer)	17 18
IN	len	the length of the indices array (integer)	19
OUT	indices	an integer array of size len, indicating category indices (array of integers)	20 21 22
int MPI_T	_category_get_categories(	int cat_index, int len, int indices[])	23 24
are contained As men as the num of the MPI added or n virtual time	ed in a particular category. A ntioned above, MPI implemen ber of variables or other cate tool information interface to new variables or categories has	RIES can be used to query which other categories category contains zero or more other categories. tations can grow the number of categories as well gories within a category. In order to allow users check quickly whether new categories have been ave been added to a category, MPI maintains a motonically increasing during the execution and is	25 26 27 28 29 30 31 32 33
MPI_T_CA	TEGORY_CHANGED(stamp)		34 35
OUT	stamp	a virtual time stamp to indicate the last change to the categories (integer)	36 37 38
int MPI_T_	_category_changed(int *st	amp)	39 40
the category	y information has not changed	e return the same timestamp, it is guaranteed that between the two calls. If the timestamp retrieved categories have been added or expanded.	41 42 43 44
Advice to users. The timestamp value is purely virtual and only intended to check for changes in the category information. It should not be used for any other purpose. (End of advice to users.)		45 46 47 48	

1 The index values returned in indices by MPI\_T\_CATEGORY\_GET\_CVARS,  $\mathbf{2}$ MPI\_T\_CATEGORY\_GET\_PVARS and MPI\_T\_CATEGORY\_GET\_CATEGORIES can be used 3 as input to MPI\_T\_CVAR\_GET\_INFO, MPI\_T\_PVAR\_GET\_INFO and 4 MPI\_T\_CATEGORY\_GET\_INFO, respectively.  $\mathbf{5}$ The user is responsible for allocating the arrays passed into the functions 6 MPI\_T\_CATEGORY\_GET\_CVARS, MPI\_T\_CATEGORY\_GET\_PVARS and 7MPI\_T\_CATEGORY\_GET\_CATEGORIES. Starting from array index 0, each function writes 8 up to len elements into the array. If the category contains more than len elements, the 9 function returns an arbitrary subset of size len. Otherwise, the entire set of elements is 10 returned in the beginning entries of the array, and any remaining array entries are not 11modified. 1213Return Codes for the MPI Tool Information Interface 14.3.9 14All functions defined as part of the MPI tool information interface return an integer error 15code (see Table 14.5) to indicate whether the function was completed successfully or was 16aborted. In the latter case the error code indicates the reason for not completing the routine. 17Such errors neither impact the execution of the MPI process nor invoke MPI error handlers. 18 The MPI process continues executing regardless of the return code from the call. The MPI 19implementation is not required to check all user-provided parameters; if a user passes invalid 20parameter values to any routine the behavior of the implementation is undefined. 21All error codes with the prefix MPI\_T\_ must be unique values and cannot overlap with 22 any other error codes or error classes returned by the MPI implementation. Further, they 23shall be treated as MPI error classes as defined in Section 8.4 on page 347 and follow the  $^{24}$ same rules and restrictions. In particular, they must satisfy: 2526 $0 = MPI_SUCCESS < MPI_T_ERR_... \le MPI_ERR_LASTCODE.$ 2728 Rationale. All MPI tool information interface functions must return error classes, 29because applications cannot portably call MPI\_ERROR\_CLASS before 30 MPI\_INIT or MPI\_INIT\_THREAD to map an arbitrary error code to an error class.  $^{31}$ (End of rationale.) 32 33 3414.3.10 Profiling Interface 35

All requirements for the profiling interfaces, as described in Section 14.2, also apply to the MPI tool information interface. All rules, guidelines, and recommendations from Section 14.2 apply equally to calls defined as part of the MPI tool information interface.

39 40

36

37

- 41 42
- 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_NOT_INITIALIZED	Interface not initialized
MPI_T_ERR_CANNOT_INIT	Interface not in the state to be initialized
Return Codes for Datatype Function	ns: MPI_T_ENUM_*
MPI_T_ERR_INVALID_INDEX	The enumeration index is invalid or has
	been deleted.
MPI_T_ERR_INVALID_ITEM	The item index queried is out of range
	(for MPI_T_ENUM_GET_ITEM only)
Return Codes for variable and cate	gory query functions: MPI_T_*_GET_INFO
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
Return Codes for Handle Functions	: MPI_T_*_{ALLOC FREE}
MPI_T_ERR_INVALID_INDEX	The variable index is invalid or has been deleted
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_OUT_OF_HANDLES	No more handles available
Return Codes for Session Functions	: MPI_T_PVAR_SESSION_*
MPI_T_ERR_OUT_OF_SESSIONS	No more sessions available
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
Return Codes for Control Variable	Access Functions:
MPI_T_CVAR_READ, WRITE	
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
Return Codes for Performance Vari	able Access and Control:
MPI_T_PVAR_{START STOP REAI	D WRITE RESET READREST}
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
MPI_T_ERR_PVAR_NO_STARTSTOP	Variable cannot be started or stopped
	(for MPI_T_PVAR_START and
	MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset
	(for MPI_T_PVAR_WRITE and
	MPI_T_PVAR_RESET)
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically
	(for MPI_T_PVAR_READRESET)
Return Codes for Category Functio	
MPI_T_ERR_INVALID_INDEX	The category index is invalid

Table 14.5: Return codes used in functions of the MPI tool information interface

# Chapter 15

 $15 \\ 16$ 

# **Deprecated Interfaces**

# 15.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI\_COMM\_CREATE\_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 17.2.7 on page 657. The language bindings are modified.

MPI\_KEYVAL\_CREATE(copy\_fn, delete\_fn, keyval, extra\_state)

24	IN	copy_fn	Copy callback function for keyval
25 26	IN	delete_fn	Delete callback function for keyval
27	OUT	keyval	key value for future access (integer)
28 29	IN	extra_state	Extra state for callback functions
30 31 32	int MPI_K		ction *copy_fn, MPI_Delete_function val, void* extra_state)
33	For this rou	utine, an interface within the	mpi_f08 module was never defined.
34 35 36 37	EXTERI	L_CREATE(COPY_FN, DELETE_ NAL COPY_FN, DELETE_FN ER KEYVAL, EXTRA_STATE, I	FN, KEYVAL, EXTRA_STATE, IERROR) ERROR
38 39 40			ten a communicator is duplicated by type MPI_Copy_function, which is defined as follows:
41 42 43 44	typedef in	void	Comm oldcomm, int keyval, *extra_state, void *attribute_val_in, *attribute_val_out, int *flag)
45 46 47 48		ran declaration for such a fun itine, an interface within the s	ction is as follows: mpi_f08 module was never defined.

```
SUBROUTINE COPY_FUNCTION(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
```

copy\_fn may be specified as MPI\_NULL\_COPY\_FN or MPI\_DUP\_FN from either C or FORTRAN; MPI\_NULL\_COPY\_FN is a function that does nothing other than returning flag = 0 and MPI\_SUCCESS. MPI\_DUP\_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute\_val\_in in attribute\_val\_out, and returns MPI\_SUCCESS. Note that MPI\_NULL\_COPY\_FN and MPI\_DUP\_FN are also deprecated.

Analogous to copy\_fn is a callback deletion function, defined as follows. The delete\_fn function is invoked when a communicator is deleted by MPI\_COMM\_FREE or when a call is made explicitly to MPI\_ATTR\_DELETE. delete\_fn should be of type MPI\_Delete\_function, which is defined as follows:

```
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi\_f08 module was never defined.

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
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 MPI\_SUCCESS. Note that MPI\_NULL\_DELETE\_FN is also deprecated.

The following function is deprecated and is superseded by MPI\_COMM\_FREE\_KEYVAL 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.

Frees the integer key value (integer)

MPI\_KEYVAL\_FREE(keyval)

keyval

INOUT

int MPI\_Keyval\_free(int \*keyval)

For this routine, an interface within the mpi\_f08 module was never defined.

```
MPI_KEYVAL_FREE(KEYVAL, IERROR)
INTEGER KEYVAL, IERROR
```

The following function is deprecated and is superseded by MPI\_COMM\_SET\_ATTR 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.

 $\mathbf{2}$ 

1 MPI\_ATTR\_PUT(comm, keyval, attribute\_val) 2 INOUT comm communicator to which attribute will be attached (han-3 dle) 4 IN key value, as returned by keyval 5MPI\_KEYVAL\_CREATE (integer) 6  $\overline{7}$ IN attribute\_val attribute value 8 9 int MPI\_Attr\_put(MPI\_Comm comm, int keyval, void\* attribute\_val) 10 For this routine, an interface within the mpi\_f08 module was never defined. 1112MPI\_ATTR\_PUT(COMM, KEYVAL, ATTRIBUTE\_VAL, IERROR) 13 INTEGER COMM, KEYVAL, ATTRIBUTE\_VAL, IERROR 14The following function is deprecated and is superseded by MPI\_COMM\_GET\_ATTR in 15MPI-2.0. The language independent definition of the deprecated function is the same as of 16the new function, except of the function name. The language bindings are modified. 1718 19MPI\_ATTR\_GET(comm, keyval, attribute\_val, flag) 2021IN comm communicator to which attribute is attached (handle) 22IN keyval key value (integer) 23OUT attribute\_val attribute value, unless flag = false $^{24}$ OUT flag true if an attribute value was extracted; false if no 2526attribute is associated with the key 2728int MPI\_Attr\_get(MPI\_Comm comm, int keyval, void \*attribute\_val, int \*flag) 29For this routine, an interface within the mpi\_f08 module was never defined. 30  $^{31}$ MPI\_ATTR\_GET(COMM, KEYVAL, ATTRIBUTE\_VAL, FLAG, IERROR) 32 INTEGER COMM, KEYVAL, ATTRIBUTE\_VAL, IERROR 33 LOGICAL FLAG 34 The following function is deprecated and is superseded by MPI\_COMM\_DELETE\_ATTR 35 in MPI-2.0. The language independent definition of the deprecated function is the same as 36 of the new function, except of the function name. The language bindings are modified. 37 38 39 MPI\_ATTR\_DELETE(comm, keyval) 40INOUT comm communicator to which attribute is attached (handle) 41 42IN keyval The key value of the deleted attribute (integer) 43 44int MPI\_Attr\_delete(MPI\_Comm comm, int keyval) 45For this routine, an interface within the mpi\_f08 module was never defined. 4647MPI\_ATTR\_DELETE(COMM, KEYVAL, IERROR) 48

INTEGER COMM, KEYVAL, IERROR

## 15.2 Deprecated since MPI-2.2

The entire set of C++ language bindings have been removed. See Chapter 16, Removed Interfaces for more information.

The following function typedefs have been deprecated and are superseded by new names. Other than the typedef names, the function signatures are exactly the same; the names were updated to match conventions of other function typedef names.

Deprecated Name	New Name
MPI_Comm_errhandler_fn	MPI_Comm_errhandler_function
MPI_File_errhandler_fn	MPI_File_errhandler_function
MPI_Win_errhandler_fn	MPI_Win_errhandler_function

# 15.3 Deprecated since MPI-4.0

The entire set of mpif.h Fortran language bindings have been deprecated.

See Chapter 17.1.4, Fortran Support Through the mpif.h Include File for more information. <sup>16</sup> ticketWG.

# Chapter 16

# **Removed Interfaces**

## 16.1 Removed MPI-1 Bindings

#### 16.1.1 Overview

The following MPI-1 bindings were deprecated as of MPI-2 and are removed in MPI-3. They may be provided by an implementation for backwards compatibility, but are not required. Removal of these bindings affects all language-specific definitions thereof. Only the language-neutral bindings are listed when possible.

#### 16.1.2 Removed MPI-1 Functions

Table 16.1 shows the removed MPI-1 functions and their replacements.

Removed	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT

Table 16.1: Removed MPI-1 functions and their replacements

#### 16.1.3 Removed MPI-1 Datatypes

Table 16.2 on page 600 shows the removed MPI-1 datatypes and their replacements.

#### 16.1.4 Removed MPI-1 Constants

Table 16.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

 $44 \\ 45$ 

	600	CHAPTER 16. REMOVED INTERFACES
1	В	emoved MPI-2 Replacement
2		PI_LB MPI_TYPE_CREATE_RESIZED
3		PI_UB MPI_TYPE_CREATE_RESIZED
4		
5		
6	Table 16.2:	Removed MPI-1 datatypes and their replacements
7		Removed MPI-1 Constants
8		C type: const int (or unnamed enum)
9		Fortran type: INTEGER
10		MPI_COMBINER_HINDEXED_INTEGER
11		MPI_COMBINER_HVECTOR_INTEGER
12		MPI_COMBINER_STRUCT_INTEGER
13		
14 15		Table 16.3: Removed MPI-1 constants
16		Table 10.5. Removed WFF-1 constants
17		
18	16.1.5 Removed MPI-1	Callback Prototypes
19	Table 16.4 shows the rem	oved MPI-1 callback prototypes and their MPI-2 replacements.
20		
21	Remo	1
22	MPI_H	andler_function MPI_Comm_errhandler_function
23		
24	Table 16.4: Rem	oved MPI-1 callback prototypes and their replacements
25		
26		
27	16.2 C++ Bindings	
28	10.2 $C++$ Dinulings	
29	The C++ bindings were	deprecated as of MPI-2.2. The C++ bindings are removed in
30		is still reserved, however, and bindings may only be provided by
31		cribed in the MPI-2.2 standard.
32	-	
33		
34		
35		
36 37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

# Chapter 17

# Language Bindings

# 17.1 Fortran Support

#### 17.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [40] + TS 29113 [41].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 17.1.7 on page 617. (End of rationale.)

MPI defines three methods of Fortran support:

- 1. USE mpi\_f08: This method is described in Section 17.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 17.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, was "frozen" at MPI-3.0, and its use is therefore not recommended. It exists only for backwards compatibility, and as a straightforward upgrade target for codes that use the deprecated mpif.h Fortran support method.
- 3. INCLUDE 'mpif.h': This method is described in Section 17.1.4. The use of the include file mpif.h [is]was strongly discouraged starting with MPI-3.0, and is deprecated starting with MPI-4.0[, because t]. 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.

<sup>39</sup> ticketWG.
 <sup>40</sup> ticketWG.
 <sup>41</sup>
 <sup>42</sup>
 <sup>43</sup>
 <sup>44</sup>
 <sup>45</sup> ticketWG.
 <sup>46</sup> ticketWG.
 <sup>47</sup> ticketWG.
 <sup>48</sup>

It exists only for backwards compatibility with legacy MPI applications and may be ticketWG. removed from a future version of MPI.

Compliant MPI-3 implementations providing a Fortran interface must 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 17.1.6 on page 614 describes restrictions if the compiler does not support all the
 needed features.

Application subroutines and functions may use either 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.

Advice to users. 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 robust and complete Fortran support. (End of advice to users.)

In a single application, it must be possible to link together routines which USE mpi\_f08, USE mpi, and INCLUDE 'mpif.h'.

The LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED is set to

23.TRUE. if all buffer choice arguments are defined in explicit interfaces with assumed-type  $^{24}$ and assumed-rank [41]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 25MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 26added to the choice buffer arguments of all nonblocking interfaces and the underlying 27Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of 28TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 29method, but not in the C header file. The values may be different for each Fortran support 30 method. All other constants and the integer values of handles must be the same for each  $^{31}$ Fortran support method.

32 Section 17.1.2 through 17.1.4 define the Fortran support methods. The Fortran in-33 terfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding full 34interface specification together with the used linker names and implications for the pro-35 filing interface. Section 17.1.6 the implementation of the MPI routines for different ver-36 sions of the Fortran standard. Section 17.1.7 summarizes major requirements for valid 37 MPI-3.0 implementations with Fortran support. Section 17.1.8 and Section 17.1.9 de-38 scribe additional functionality that is part of the Fortran support. MPI\_F\_SYNC\_REG 39 is needed for one of the methods to prevent register optimization problems. A set of func-40 tions provides additional support for Fortran intrinsic numeric types, including parameter-41 ized types: MPI\_SIZEOF, MPI\_TYPE\_MATCH\_SIZE, MPI\_TYPE\_CREATE\_F90\_INTEGER, 42MPI\_TYPE\_CREATE\_F90\_REAL and MPI\_TYPE\_CREATE\_F90\_COMPLEX. In the context 43of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 44parameters. Sections 17.1.10 through 17.1.19 give an overview and details on known prob-45lems when using Fortran together with MPI; Section 17.1.20 compares the Fortran problems 46with those in C. 47

48

1

 $\mathbf{2}$ 

3 4

5 6

7 8

9

15

16

17

18

19 20

21

### 17.1.2 Fortran Support Through the mpi\_f08 Module

08 Module

An MPI implementation providing a Fortran interface must provide a module named mpi\_f08 that can be used in a Fortran program. Section 17.1.6 on page 614 describes restrictions if the compiler does not support all the needed features. Within all MPI function specifications, the first of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not TYPE(\*), with the following exception:

Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the mpi module, see Section 17.1.3 on page 605.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from mpif.h or the mpi module to the mpi\_f08 module. (End of advice to users.)

- Define all MPI handles with uniquely named handle types (instead of INTEGER handles, as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators .EQ. and .NE. to allow the comparison of these MPI handles with .EQ., .NE., == and /=.
- Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations, and set the LOGICAL compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113). See Section 17.1.6 on page 614 for older compiler versions.
- Set the LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., TYPE(\*), DIMENSION(..) in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

*Rationale.* In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TS 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 TS 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 17.1.6 on page 614 for details.

 $\mathbf{5}$ 

 $^{31}$ 

- 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 [40], 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 17.1.3 on page 605. (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\_COMM\_NULL\_COPY\_FN).

*Rationale.* For user-defined callback functions (e.g., COMM\_COPY\_ATTR\_FUNCTION) and their predefined callbacks (e.g., MPI\_COMM\_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.*)

The MPI Fortran bindings in the mpi\_f08 module are designed based on the Fortran 2008 standard [40] together with the Technical Specification "TS 29113 Further Interoperability with C" [41] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [41], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the mpi\_f08 module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g.,

scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., REAL, CHARACTER*5, CHARACTER*(*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi_f08 module).	1 2 3 4 5 6 7 8
The INTERFACE construct in combination with BIND(C) allows the implementation of the Fortran mpi_f08 interface with a single set of portable wrapper routines written in C, which supports all desired features in the mpi_f08 interface. TS 29113 also has a provision for OPTIONAL arguments in BIND(C) interfaces.	9 10 11 12
A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.	12 13 14 15 16
The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. ( <i>End of rationale.</i> )	17 18 19
17.1.3 Fortran Support Through the mpi Module An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran program. The mpi module has been "frozen" as of MPI-3.0, meaning that no new functionality will be added to the module after MPI-3.0.	20 21 22 23 ticketWG. 24
Advice to users. The mpi module provides a straightforward upgrade path for codes that currently include mpif.h. Users should be aware, however, that the mpi_f08 module contains the most modern Fortran MPI support, and will likely have a longer life than the mpi module. Users may upgrade to the mpi module in the short term, but should also investigate upgrading to the mpi_f08 module in the longer term. (End of advice to users.)	25 26 27 28 29 30 31
Within all [MPI]MPI-3.0 function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:	<sup>32</sup> ticketWG. <sup>33</sup>
• Define all named [MPI]MPI-3.0 constants	35 ticketWG.
• Declare [MPI]MPI-3.0 functions that return a value.	$^{36}_{_{37}}$ ticketWG.
• 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.	38 39 40
• Define all [MPI]MPI-3.0 handles as type INTEGER.	$^{41}_{42}$ ticketWG.
• Define the derived type MPI_Status and all named handle types that are used in the mpi_f08 module. For these named handle types, overload the operators .EQ. and .NE. to allow handle comparison via the .EQ., .NE., == and /= operators.	43 44 45 46
Rationale. They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (End of rationale.)	47 48

- 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.
- Set the LOGICAL 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 TS 29113), otherwise to .FALSE..

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 constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another 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 copy-in/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 17.1.12 on page 630 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 TS 29113 language feature. See Section 17.1.6 on page 614 for further details.
  - Set the LOGICAL 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.. When MPI\_SUBARRAYS\_SUPPORTED is defined as .TRUE., non-contiguous sub-arrays 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 TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in non-blocking calls may be disallowed. See Section 17.1.6 on page 614 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

<sup>45</sup> Advice to implementors. The appropriate INTENT may be different from what is given <sup>46</sup> in the MPI language-neutral bindings. Implementations must choose INTENT so that <sup>47</sup> the function adheres to the MPI standard, e.g., by defining the INTENT as provided in <sup>48</sup> the mpi\_f08 bindings. (End of advice to implementors.)

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

1

*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 was changed in several places in MPI-2. For instance, MPI\_IN\_PLACE changes the intent of an OUT argument to be INOUT. (*End of rationale.*)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than 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-TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include mpif.h may not expect that 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 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.)

### 17.1.4 Fortran Support Through the mpif.h Include File

The use of the mpif.h include file is [strongly discouraged]deprecated and may be [deprecated in]deleted from a future version of MPI.

Advice to users. Users are strongly encouraged to upgrade codes that include mpif.h to use the mpi module; the process should be fairly straightforward (see the next Advice to Users, below). Note, however, that the mpi module support in MPI is "frozen" and will not be expanded beyond MPI-3.0. For a longer-term solution, the mpi\_f08 module provides more modern Fortran features and will continue to be expanded beyond MPI-3.0. (End of advice to users.)

An MPI implementation providing a Fortran interface must provide 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:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Define all handles as INTEGER.
- Be valid and equivalent for both fixed and free source form.

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).

 $\mathbf{2}$ 

 $46 \\ 47$ 

1 2 3 4	• Set the LOGICAL 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
5 6 7	Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is strongly encouraged for the following reasons:
8	• Most mpif.h implementations do not include compile-time argument checking.
10 11	• Therefore, many bugs in MPI applications remain undetected at compile-time, such as:
12	
13	<ul> <li>Missing ierror as last argument in most Fortran bindings.</li> <li>Declaration of a status as an INTEGER variable instead of an INTEGER array</li> </ul>
14	with size MPI_STATUS_SIZE.
15	- Incorrect argument positions; e.g., interchanging the <b>count</b> and
16 17	datatype arguments.
18 19	<ul> <li>Passing incorrect MPI handles; e.g., passing a datatype instead of a commu- nicator.</li> </ul>
20	• The migration from mpif.h to the mpi module should be relatively straightfor-
21	ward (i.e., substituting include 'mpif.h' after an implicit statement by use
22	mpi before that implicit statement) as long as the application syntax is correct.
23	• Migrating portable and correctly written applications to the mpi module is not
24 25	expected to be difficult. No compile or runtime problems should occur because
26	an mpif.h include file was always allowed to provide explicit Fortran interfaces.
27	(End of advice to users.)
ticketWG. 28	r
29	
30	Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to
31 32	retain strong backward compatibility. Internally, mpif.h and the mpi module may be
33	implemented so that essentially the same library implementation of the MPI routines $\left( E_{i} \right)$
34	can be used. (End of rationale.)
35	]
36	Advise to implementance. To make <b>which is</b> competible with both fixed and free sources
37 38	Advice to implementors. To make mpif.h compatible with both fixed- and free-source forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-
39	form line length, it is recommended that the requirement of usability in free and fixed
40	source form applications be met by constructing mpif.h without any continuation
41	lines. This should be possible because mpif.h may contain only declarations, and
42	because common block declarations can be split among several lines. The argument
43	names may need to be shortened to keep the SUBROUTINE statement within the allowed $72 - 6 = 66$ abaracters, a.g.
44 45	72 - 6 = 66 characters, e.g.,
46	INTERFACE
47	SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
48	! dummy argument declarations

This line has 65 characters and is the longest in MPI-3.0. 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:

9
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
12

This would need a line length of 73 characters, i.e., the C routine name would need to be shortened by 7 characters to stay within the available 66 characters. 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 TS 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.*)

#### 17.1.5 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 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 foofoo\_\_. In the case of BIND(C,NAME='...'), the linker name is directly defined through the external name given by the string.

The following rules for linker names apply:

• With the Fortran mpi\_f08 module, if MPI\_SUBARRAYS\_SUPPORTED equals .TRUE.:

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:

```
INTERFACE
SUBROUTINE MPI_Send(...) BIND(C,NAME='MPI_Send_f08')
```

• With the Fortran mpi\_f08 module, if MPI\_SUBARRAYS\_SUPPORTED equals .FALSE. (i.e., with a preliminary implementation of this module without TS 29113):

The linker name of each routine is defined through the linker name mapping of the Fortran compiler for the name defined when subarrays are supported. For example, MPI\_Send\_f08 may be mapped to mpi\_send\_f08\_. Example:

INTERFACE MPI\_Send
SUBROUTINE MPI\_Send\_f08(...)

 $\mathbf{2}$ 

 $^{24}$ 

1	• With the Fortran mpi module or mpif.h include file, if MPI_SUBARRAYS_SUPPORTED
2	equals .FALSE.:
3	The linker name of each routine is defined through the linker-name mapping of the
4	Fortran compiler. For example, MPI_SEND may be mapped to mpi_send Example:
5	
6	INTERFACE
7	SUBROUTINE MPI_SEND()
8	
9	• With the Fortran mpi module or mpif.h include file, if MPI_SUBARRAYS_SUPPORTED
10	equals .TRUE .:
11	The Fortran binding must use BIND(C) interfaces with an interface name identical to
12	the language independent name, e.g., MPI_SEND. The linker name is a combination
13	of the C name and an _f suffix, e.g., MPI_Send_f. Prototype example:
14	of the O fiame and an _1 sumx, e.g., in 1_Send_1. I fototype example.
15	INTERFACE
16	SUBROUTINE MPI_SEND() BIND(C,NAME='MPI_Send\_f')
17	SUBRUUTINE MFI_SEND() BIND(C,NAME- MFI_Send(_I)
18	If the support of subarrays is different for the mpi module and the mpif.h include file,
19	then both linker-name methods can be used in the same application. If the application also
20	uses the mpi_f08 module and was compiled with this module partially before and after the
21	subarrays were supported, then all four interfaces are used within the same application.
22	subarrays were supported, then an four interfaces are used within the same application.
23	Rationale. After a compiler provides the facilities from TS 29113, i.e., TYPE(*),
24	DIMENSION(), it is possible to change the bindings within a Fortran support method
25	to support subarrays without recompiling the complete application. Of course, only
26	recompiled routines can benefit from the added facilities. There is no binary compat-
27	ibility conflict because each interface uses its own linker names and all interfaces use
28	the same constants and type definitions. (End of rationale.)
29 30	
31	A user-written or middleware profiling routine that is written according to the same
32	binding rules will have the same linker name, and therefore, can interpose itself as the MPI
33	library routine. The profiling routine can internally call the matching PMPI routine with any
34	of its existing bindings, except for routines that have callback routine dummy arguments.
35	In this case, the profiling software must use the same Fortran support method as used in
36	the calling application program, because the C, mpi_f08 and mpi callback prototypes are
37	different.
38	Advice to users. This advice is mainly for tool writers. Even if an MPI library
39	supports subarrays in all three Fortran support methods, a portable profiling layer
40	should also provide the two interfaces for MPI_SUBARRAYS_SUPPORTED==.FALSE.
40	to support older binary user routines that were compiled before TS 29113 level support
41	was availabile.
43	
43	If a user application calls MPI_SEND, then the chosen Fortran support method to- gether with the MPI implementation decision about MPI SUBARRAYS SUPPORTED
45	gether with the MPI implementation decision about MPI_SUBARRAYS_SUPPORTED
46	imply, to which linker name the compiler will translate this call, i.e., whether the application calls and com MPL Sond for maj cond for
47	application calls mpi_send, or MPI_Send_f, or mpi_send_f08, or MPI_Send_f08.
48	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:
	MPI implementation, then it should provide all four routines. For example:

```
SUBROUTINE MPI_SEND(...) BIND(C,NAME='MPI_Send_f')
   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.)

If an implementation provides in a first step two sets of Advice to implementors. routines, one for the mpi module and mpif.h, and the other for the mpi\_f08 module, and both sets without TS 29113, i.e., MPI\_SUBARRAYS\_SUPPORTED equals .FALSE., and the implementor wants to add a TS 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 indicating 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 25routines, 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 27rule about routine grouping is fulfilled. The implementation of routines with ASYNCHRONOUS 28 choice buffers depends on the rules for the provided Fortran support method and language 29 level of the underlying compiler. Predefined callback routines for the mpi\_f08 module 30 must be implemented with BIND(C) interfaces, and for the mpi module and mpif.h without 31BIND(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 be 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 must not be available through this scheme. Peer routines and their routine groups:

1  $\mathbf{2}$ 

3

4

5

6

7 8

9

10

11

12

13

14

1516

17

18

19

20

21

22

23

 $^{24}$ 

26

32

33

34

35

36

37

1		API_ALLOC_MEM, MPI_WIN_ALLOCATE,
2		1PI_WIN_SHARED_ALLOCATE, and
3		IPI_WIN_SHARED_QUERY.
4		Only this routine is in this group.
5		Only this routine is in this group.
6		Il routines with choice buffer arguments that
7		re not declared as ASYNCHRONOUS within the
8		pi_f08 module.
9		Il routines with choice buffer arguments
10		hat are declared as ASYNCHRONOUS within the
11		pi_f08 module.
12		Only this routine is in this group.
13		Only this routine is in this group.
14		Il other routines with callback function argu-
15		nents.
16		Il predefined callback routines.
17	MPI_COMM_RANK A	Il other MPI routines.
18	Advise to implementary Pomor	ed interfaces (see Chapter $16$ ) are in the same routine
19	*	acement functions. ( <i>End of advice to implementors.</i> )
20	group as then corresponding repr	acement functions. (Ena of advice to implementors.)
21	Additionally, four C preprocessor	macros are available in mpi.h for each routine group.
22	· · · · · · · · · · · · · · · · · · ·	utine name written as in the list above and appended
23	with one of the following suffixes and r	
24	-	et to 1 if the BIND(C) linker name with the
25	-	B is available for all routines within this group
26		_f08), otherwise it is set to 0.
27	· - ·	et to 1 if the Fortran linker name with the
28	•	is available for all routines within this group
29		f08), otherwise it is set to 0.
30		et to 1 if the BIND(C) linker name with the
31 32	linker suffix _f i	s available for all routines within this group
		_f), otherwise it is set to 0.
33 34	_mpi_BIND_F The macro is se	et to 1 if the Fortran linker name without
	a linker suffix is	available for all routines within this group
35 36	(e.g., mpi_send_	_), otherwise it is set to 0.
37	For example	
38		
39		
40	<pre>#define MPI_SEND_mpi_f08_BI #define MPI_SEND_mpi_f08_BI</pre>	
41	#define MPI_SEND_mpi_BIND_C	
42	#define MPI_SEND_mpi_BIND_C #define MPI_SEND_mpi_BIND_F	
43	#deline Wri_Srwp_mbi_piwp_r	1
44	<pre>#define MPI_ISEND_mpi_f08_B</pre>	IND_C 1
45	#define MPI_ISEND_mpi_f08_B	
46	#define MPI_ISEND_mpi_BIND_	
47	#define MPI_ISEND_mpi_BIND_ #define MPI_ISEND_mpi_BIND_	
48	-	1 I
	•••	

```
#define MPI_COMM_DUP_FN_mpi_f08_BIND_C 1
#define MPI_COMM_DUP_FN_mpi_f08_BIND_F 0
#define MPI_COMM_DUP_FN_mpi_BIND_C 0
#define MPI_COMM_DUP_FN_mpi_BIND_F 1
```

shows, that

- the routines in the MPI\_SEND group are only available through their Fortran linker names (e.g., mpi\_send\_f08\_, mpi\_send\_\_, mpi\_recv\_f08\_, mpi\_recv\_\_, ...),
- the routines in the MPI\_ISEND group are available with all four interfaces: the MPI library, the mpi\_f08 and mpi modules (that provide the TS 29113 quality), and this MPI library supports application routines that are compiled with an older MPI library version with \_BIND\_C set to 0 and \_BIND\_F set to 1.

For the predefined callbacks, there is no choice, because the interfaces must fit to the callback function prototypes which are BIND(C) based for mpi\_f08 and without BIND(C) for the mpi module and mpif.h.

Advice to implementors. If all following conditions are fulfilled (which is the case for most compilers):

- the handles in the mpi\_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then for the routine groups, the implementor may use the same internal routine implementations for all Fortran support methods but with several different linker names. For TS 29113 quality, new routines are needed only for the routine group of MPI\_ISEND. Typical settings for \_mpi\_f08\_BIND\_C / \_mpi\_f08\_BIND\_F / \_mpi\_BIND\_C / \_mpi\_BIND\_F may be:

	Without TS 29113	Upgrade to TS 29113	Upgrade for strided data optimization	New impl. with TS 29113	38 39
MPI_ALLOC_MEM	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	40
MPI_FREE_MEM	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	41
MPI_GET_ADDRESS	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	42
MPI_SEND	0/1/0/1	0/1/0/1	1/1/1/1	1/0/1/0	43
MPI_ISEND	0/1/0/1	1/1/1/1	1/1/1/1	1/0/1/0	44
MPI_OP_CREATE	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	45
MPI_REGISTER_DATAREP	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	46
MPI_COMM_KEYVAL_CREATE	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	47
MPI_COMM_DUP_FN	1/0/0/1	1/0/0/1	1/0/0/1	1/0/0/1	48
MPI_COMM_RANK	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0	48

 $^{24}$ 

$\frac{1}{2}$	(End of advice to implementors.)
3	17.1.6 MPI for Different Fortran Standard Versions
4 5 6	This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.
7 8	• For Fortran 77 with some extensions:
9	- MPI identifiers may be up to 30 characters (31 with the profiling interface).
10 11	– MPI identifiers may contain underscores after the first character.
11 12 13	<ul> <li>An MPI subroutine with a choice argument may be called with different argument types.</li> </ul>
14 15	<ul> <li>Although not required by the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.</li> </ul>
16 17 18 19	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 does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem
20	is solved with $MPI_GET_ADDRESS$ , but not for Fortran 77.)
21 22 23	• For Fortran 90: The major additional features that are needed from Fortran 90 are:
24	- The MODULE and INTERFACE concept.
25	- The KIND= and SELECTEDKIND concept.
26	- Fortran derived TYPEs and the SEQUENCE attribute.
27 28	- The OPTIONAL attribute for dummy arguments.
29 30	<ul> <li>Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.</li> </ul>
31	With these features, $MPI-1.1 - MPI-2.2$ can be implemented without restrictions.
32 33	MPI-3.0 can be implemented with some restrictions. The Fortran support methods
34	are abbreviated with $S1 = \text{the mpi}_f08 \text{ module}$ , $S2 = \text{the mpi} \text{ module}$ , and $S3 = \text{the mpi}$
35	mpif.f include file. If not stated otherwise, restrictions exist for each method which
36	prevent implementing the complete semantics of MPI-3.0.
37	– MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-
38 39	contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,
40	or split-collective I/O.
41 42	<ul> <li>S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa- tion is possible.</li> </ul>
43	- In this preliminary interface of S1, the following changes are necessary:
44	* The routines are not BIND(C).
45 46	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
40 47	like !\$PRAGMA IGNORE_TKR.
48	* The ASYNCHRONOUS attribute is omitted.

- \* **PROCEDURE(...)** callback declarations are substituted by **EXTERNAL**.
- The linker names are specified in Section 17.1.5 on page 609.
- Due to the rules specified in Section 17.1.5 on page 609, choice buffer declarations should be implemented only with non-standardized extensions like !\$PRAGMA IGNORE\_TKR (as long as F2008+TS 29113 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 17.1.11 on page 629). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 17.1.15 on page 633.

Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, 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 related to TYPE(MPI\_Status) (see Section 17.2.5 on page 652) are also not available in the mpi module and mpif.h.
- For Fortran 95: The quality of the MPI interface and the restrictions are the same as with Fortran 90.

• For Fortran 2003:

The major features that are needed from Fortran 2003 are:

- Interoperability with C, i.e.,
  - \* BIND(C, NAME='...') interfaces.
  - \* BIND(C) derived types.
  - \* The ISO\_C\_BINDING intrinsic type C\_PTR and routine C\_F\_POINTER.
- The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.
- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O.
   This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.

With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:

1 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 2	<ul> <li>The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR</li> </ul>
3	and an $INTEGER(KIND=MPI_ADDRESS_KIND)$ argument must both map to a
4	void * argument.
5 6	MPI-3.0 can be implemented with the following restrictions:
7	- MPI_SUBARRAYS_SUPPORTED equals .FALSE
8 9	- For S1, only a preliminary implementation is possible. The following changes are
10	necessary:
11	* The routines are not BIND(C).
12 13	* TYPE(*), DIMENSION() is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.
14	- The linker names are specified in Section 17.1.5 on page 609.
15	- With S1, the ASYNCHRONOUS is required as specified in the second Fortran inter-
16	faces. With S2 and S3 the implementation can also add this attribute if explicit
17 18	interfaces are used.
19	- The ASYNCHRONOUS Fortran attribute can be used in applications to try to protect
20	buffers in nonblocking MPI calls, but the protection can work only if the compiler
21	is able to protect a synchronous Fortran $\mathrm{I}/\mathrm{O}$ and makes no difference between such
22	asynchronous Fortran I/O and $MPI$ communication.
23	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
24	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
25	be used only for Fortran types that are C compatible.
26	- The same restriction as for Fortran 90 applies if non-standardized extensions like
27 28	<b>!\$PRAGMA IGNORE_TKR</b> are not available.
29	For Fortran $2008 + TS 29113$ and later and
30	For Fortran 2003 + TS 29113:
31	The major feature that are needed from TS 29113 are:
32 33	- TYPE(*), DIMENSION() is available.
34	- The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com-
35	munication.
36	
37	- OPTIONAL dummy arguments are allowed in combination with BIND(C) interfaces.
38	- CHARACTER(LEN=*) dummy arguments are allowed in combination with BIND(C)
39	interfaces.
40	$-$ The array dummy argument of the <code>ISO_C_BINDING</code> intrinsic <code>C_F_POINTER</code> is not
41	restricted to Fortran types for which a corresponding type in C exists.
42 43	Using these features, $MPI-3.0$ can be implemented without any restrictions.
44	- With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS at-
45	tribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR)
46	binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
47	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
48	be used for any Fortran type.

<ul> <li>With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation dependent. A high quality implementation will also provide</li> <li>MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.</li> </ul>
<ul> <li>If non-standardized extensions like !\$PRAGMA IGNORE_TKR are not available then S2 must be implemented with TYPE(*), DIMENSION().</li> </ul>
Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice argument may be implemented with an explicit interface using compiler directives, for example:
INTERFACE
SUBROUTINE MPI(buf,)
!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf
!\$PRAGMA IGNORE_TKR buf
!DIR\$ IGNORE_TKR buf
!IBM* IGNORE_TKR buf
REAL, DIMENSION(*) :: buf
! declarations of the other arguments
END SUBROUTINE
END INTERFACE

(End of advice to implementors.)

## 17.1.7 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 17.1.11 through 17.1.19 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:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 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 an MPI library that implements the mpi\_f08 module with MPI\_SUBARRAYS\_SUPPORTED 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 17.1.12 on page 630 for more details.

 $^{24}$ 

 $^{31}$ 

1 2 3 4	• 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
5 6 7 8 9	• 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.
10 11 12 13	• 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.
14 15 16	• The Fortran compiler shall not provide TYPE(*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.
17 18 19 20 21 22 23	The following rules are required at least as long as the compiler does not provide the ex- tension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==.FALSE Observation of these rules by the MPI application developer is especially recomended for backward com- patibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows:
24 25 26 27	• 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 641 and Section 17.1.8 on page 619, and DD on page 642) solve the problems described in Section 17.1.17 on page 636.
28 29 30 31 32	• The problems with temporary data movement (described in detail in Section 17.1.18 on page 643) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
33 34 35 36 37	• Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 17.1.19 on page 646) are resolved <b>without</b> 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 invoking MPI procedures.
38 39 40	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 if mpif.h uses explicit interfaces.
41 42 43 44 45 46	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 TS 29113 [41]. Some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (End of advice to implementors.)
47 48	Further requirements apply if 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 TS 29113 (e.g., CHARACTER\*(\*)).
- OPTIONAL dummy arguments are also valid within BIND(C) interfaces. This requirement is fulfilled if TS 29113 is fully supported by the compiler.

## 17.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 17.1.17 on page 636, 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 has no executable statements. 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 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. (End of advice to implementors.)

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. (End of rationale.)

 $\mathbf{5}$ 

 $^{31}$ 

 $45 \\ 46$ 

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 be called if MPI\_ASYNC\_PROTECTS\_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (*End of advice to users.*)

# 17.1.9 Additional Support for Fortran Numeric Intrinsic Types

<sup>10</sup> MPI provides a small number of named datatypes that correspond to named intrinsic types <sup>11</sup> supported by C and Fortran. These include MPI\_INTEGER, MPI\_REAL, MPI\_INT,

<sup>12</sup> MPI\_DOUBLE, etc., as well as the optional types MPI\_REAL4, MPI\_REAL8, etc. There is a <sup>13</sup> one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These 14types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and 15CHARACTER) with an optional integer KIND parameter that selects from among one or more 16variants. The specific meaning of different KIND values themselves are implementation 17dependent and not specified by the language. Fortran provides the KIND selection functions 18 selected\_real\_kind for REAL and COMPLEX types, and selected\_int\_kind for INTEGER 19types that allow users to declare variables with a minimum precision or number of digits. 20These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 21INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL 22and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 23**PRECISION** variables are of intrinsic type **REAL** with a non-default KIND. The following two  $^{24}$ declarations are equivalent: 25

double precision x real(KIND(0.0d0)) x

29MPI provides two orthogonal methods for handling communication buffers of numeric 30intrinsic types. The first method (see the following section) can be used when variables have  $^{31}$ been declared in a portable way — using default KIND or using KIND parameters obtained 32 with the selected\_int\_kind or selected\_real\_kind functions. With this method, MPI 33 automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation 34conversion in heterogeneous environments. The second method (see "Support for size-35 specific MPI Datatypes" on page 624) gives the user complete control over communication 36 by exposing machine representations.

37 38

26

27

28

# Parameterized Datatypes with Specified Precision and Exponent Range

<sup>39</sup> MPI provides named datatypes corresponding to standard Fortran 77 numeric types:

<sup>41</sup> MPI\_INTEGER, MPI\_COMPLEX, MPI\_REAL, MPI\_DOUBLE\_PRECISION and

MPI\_DOUBLE\_COMPLEX. MPI automatically selects the correct data size and provides rep resentation 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

1

2

3

4

5

6

7 8

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)

<pre>int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)</pre>
<pre>MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C) INTEGER, INTENT(IN) :: p, r</pre>
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MDT TYDE CREATE EQA REAT (D R NEWTYDE TERRAR)

### MPI\_TYPE\_CREATE\_F90\_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected\_real\_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected\_real\_kind(p, r) (but not both). Analogously, either p or r may be set to MPI\_UNDEFINED. In communication, an MPI datatype A returned by MPI\_TYPE\_CREATE\_F90\_REAL matches a datatype B if and only if B was returned by MPI\_TYPE\_CREATE\_F90\_REAL called with the same values for p and r or B is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 624.

It is erroneous to supply values for  $\boldsymbol{p}$  and  $\boldsymbol{r}$  not supported by the compiler.

 $\mathbf{2}$ 

 $\overline{7}$ 

```
1
     MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)
2
       IN
                                             precision, in decimal digits (integer)
                 р
3
       IN
                 r
                                             decimal exponent range (integer)
4
5
       OUT
                                             the requested MPI datatype (handle)
                 newtype
6
7
     int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
8
     MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
9
          INTEGER, INTENT(IN) :: p, r
10
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
11
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
12
13
     MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
14
          INTEGER P, R, NEWTYPE, IERROR
15
         This function returns a predefined MPI datatype that matches a
16
     COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from
17
     calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set
18
     to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to
19
     the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions
20
     on using the returned datatype with the "external32" data representation are given on
21
     page 624.
22
         It is erroneous to supply values for p and r not supported by the compiler.
23
^{24}
25
     MPI_TYPE_CREATE_F90_INTEGER(r, newtype)
26
       IN
                                             decimal exponent range, i.e., number of decimal digits
27
                 r
                                             (integer)
28
29
       OUT
                 newtype
                                             the requested MPI datatype (handle)
30
^{31}
     int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
32
33
     MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
34
          INTEGER, INTENT(IN) :: r
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
35
          INTEGER, OPTIONAL, INTENT(OUT) ::
36
                                                  ierror
37
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
38
          INTEGER R, NEWTYPE, IERROR
39
40
          This function returns a predefined MPI datatype that matches a INTEGER variable of
41
     KIND selected_int_kind(r). Matching rules for datatypes created by this function are
42
     analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
     Restrictions on using the returned datatype with the "external32" data representation are
43
44
     given on page 624.
45
         It is erroneous to supply a value for r that is not supported by the compiler.
46
         Example:
47
         integer
                        longtype, quadtype
48
```

```
integer, parameter :: long = selected_int_kind(15)
integer(long) ii(10)
real(selected_real_kind(30)) x(10)
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
...
call MPI_SEND(ii, 10, longtype, ...)
call MPI_SEND(x, 10, quadtype, ...)
```

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\_XXX 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\_XXX with the same combination of (XXX,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, 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\_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,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 13.7.2 on page 534) or user-defined (Section 13.7.3 on page 535) 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 13.7.2 on page 534.

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 <sup>48</sup>

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 

1values are represented by one of three IEEE formats. These are the IEEE "Single," "Dou- $\mathbf{2}$ ble," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. 3 For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 4 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the  $\mathbf{5}$ "Double" format. 6 The external 32 representations of the datatypes returned by 7MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER are given by the following rules. 8 For MPI\_TYPE\_CREATE\_F90\_REAL: 9 (p > 33) or (r > 4931) then external32 representation if 10 is undefined 11 307) then external32\_size = 16 else if (p > 15) or (r > 12else if (p > 6) or (r >37) then external32\_size = 8 13  $external32_size = 4$ else 1415For MPI\_TYPE\_CREATE\_F90\_COMPLEX: twice the size as for 16MPI\_TYPE\_CREATE\_F90\_REAL. 17For MPI\_TYPE\_CREATE\_F90\_INTEGER: 18 19if (r > 38) then external32 representation is undefined 20else if (r > 18) then external32\_size = 16 21else if (r > 9) then external32\_size = 8 22else if (r > 4) then external32\_size = 4 23else if (r > 2) then external32\_size = 2  $^{24}$  $external32_size = 1$ else 25

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI\_PACK\_EXTERNAL, MPI\_UNPACK\_EXTERNAL, and many MPI\_FILE functions, when the "external32" data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

32 33

Support for Size-specific MPI Datatypes

<sup>34</sup> MPI provides named datatypes corresponding to optional Fortran 77 numeric types that <sup>35</sup> contain explicit byte lengths — MPI\_REAL4, MPI\_INTEGER8, etc. This section describes a <sup>36</sup> 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 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:

- 44 MPI\_REAL4
- 45 MPI\_REAL8
- 46 MPI\_REAL16
- 47 MPI\_COMPLEX8
- 48 MPI\_COMPLEX16

MPI_COMPLEX32	1
MPI_INTEGER1	2
MPI_INTEGER2	3
MPI_INTEGER4	4
MPI_INTEGER8	5
MPI_INTEGER16	6
	7
One datatype is required for each representation supported by the compiler. To be backward	8
compatible with the interpretation of these types in MPI-1, we assume that the nonstandard	9

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)
    IN x a Fortran variable of numeric intrinsic type (choice)
    OUT size size of machine representation of that type (integer)

MPI_Sizeof(x, size, ierror) BIND(C)
    TYPE(*), DIMENSION(..) :: x
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR</type>
```

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C size of operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (End of advice to users.)

*Rationale.* This function is not available in other languages because it would not be useful. (*End of rationale.*)

## MPI\_TYPE\_MATCH\_SIZE(typeclass, size, datatype)

IN	typeclass	generic type specifier (integer)	43
IN		· · · · · · · · · · · · · · · · · · ·	44
IIN	SIZE	size, in bytes, of representation (integer)	45
OUT	datatype	datatype with correct type, size (handle)	46
			47

int MPI\_Type\_match\_size(int typeclass, int size, MPI\_Datatype \*datatype)

 $^{24}$ 

```
1
     MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C)
\mathbf{2}
          INTEGER, INTENT(IN) :: typeclass, size
3
          TYPE(MPI_Datatype), INTENT(OUT) :: datatype
4
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
5
     MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
6
          INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
7
8
          typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and
9
     MPI_TYPECLASS_COMPLEX, corresponding to the desired typeclass. The function returns
10
     an MPI datatype matching a local variable of type (typeclass, size).
11
          This function returns a reference (handle) to one of the predefined named datatypes, not
12
     a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a
13
     size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF
14
     in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find
15
     a suitable datatype. In C, one can use the C function sizeof(), instead of MPI_SIZEOF.
16
     In addition, for variables of default kind the variable's size can be computed by a call to
17
     MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not
18
     supported by the compiler.
19
           Rationale. This is a convenience function. Without it, it can be tedious to find the
20
           correct named type. See note to implementors below. (End of rationale.)
21
22
           Advice to implementors. This function could be implemented as a series of tests.
23
24
           int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
25
           {
26
             switch(typeclass) {
27
                 case MPI_TYPECLASS_REAL: switch(size) {
28
                    case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
29
                    case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
30
                    default: error(...);
31
                 }
32
                 case MPI_TYPECLASS_INTEGER: switch(size) {
33
                     case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
34
                     case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
35
                     default: error(...);
36
                 }
37
                 ... etc. ...
38
              }
39
40
              return MPI_SUCCESS;
41
           }
42
43
           (End of advice to implementors.)
44
45
     Communication With Size-specific Types
46
47
     The usual type matching rules apply to size-specific datatypes: a value sent with datatype
48
     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 communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
33
real(selected_real_kind(5)) x(100)
                                                                               34
call MPI_SIZEOF(x, size, ierror)
                                                                               35
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
                                                                               36
                                                                               37
if (myrank .eq. 0) then
                                                                               38
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                               &
                                                                               39
                       MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                               &
                                                                               40
                       MPI_INFO_NULL, fh, ierror)
                                                                               41
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                               42
                           MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                               43
                                                                               44
   call MPI_FILE_CLOSE(fh, ierror)
                                                                               45
endif
                                                                               46
                                                                               47
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
                                                                               48
```

```
1
           if (myrank .eq. 1) then
2
               call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY, &
3
                               MPI_INFO_NULL, fh, ierror)
4
               call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
5
                                         MPI_INFO_NULL, ierror)
6
               call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
7
               call MPI_FILE_CLOSE(fh, ierror)
8
           endif
9
10
11
           If processes 0 and 1 are on different machines, this code may not work as expected if
12
           the size is different on the two machines. (End of advice to users.)
13
14
     17.1.10
              Problems With Fortran Bindings for MPI
15
     This section discusses a number of problems that may arise when using MPI in a Fortran
16
     program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It
17
     is intended to clarify, not add to, this standard.
18
          As noted in the original MPI specification, the interface violates the Fortran standard
19
     in several ways. While these may cause few problems for Fortran 77 programs, they become
20
     more significant for Fortran 90 programs, so that users must exercise care when using new
21
     Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most
22
     violations are resolved, and this is hinted at in an addendum to each item. The violations
23
     were originally adopted and have been retained because they are important for the usability
^{24}
     of MPI. The rest of this section describes the potential problems in detail.
25
          The following MPI features are inconsistent with Fortran 90 and Fortran 77.
26
27
        1. An MPI subroutine with a choice argument may be called with different argument
28
           types. When using the mpi_f08 module together with a compiler that supports For-
29
           tran 2008 + TS 29113, this problem is resolved.
30
31
        2. An MPI subroutine with an assumed-size dummy argument may be passed an actual
32
           scalar argument. This is only solved for choice buffers through the use of
33
           DIMENSION(..).
34
        3. Nonblocking and split-collective MPI routines assume that actual arguments are passed
35
           by address or descriptor and that arguments and the associated data are not copied
36
           on entrance to or exit from the subroutine. This problem is solved with the use of the
37
           ASYNCHRONOUS attribute.
38
39
        4. An MPI implementation may read or modify user data (e.g., communication buffers
40
           used by nonblocking communications) concurrently with a user program that is ex-
41
           ecuting outside of MPI calls. This problem is resolved by relying on the extended
42
           semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
43
44
        5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE,
45
           MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE,
46
           MPI_UNWEIGHTED, MPI_WEIGHTS_EMPTY, MPI_ARGV_NULL, and MPI_ARGVS_NULL
47
           are not ordinary Fortran constants and require a special implementation. See Sec-
48
           tion 2.5.4 on page 15 for more information.
```

6. The memory allocation routine MPI\_ALLOC\_MEM cannot be used from 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. A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 on page 17 and Section 4.1.1 on page 85 for more information.

Sections 17.1.11 through 17.1.19 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 17.1.7 on page 617.

## 17.1.11 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 17.1.6 on page 614). In C, the use of void\* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 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.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5) 44

real x(5) 45

... 46

call mpi_send(x, 5, MPI_REAL, ...) 47

call mpi_send(i, 5, MPI_INTEGER, ...) 48
```

 $^{24}$ 

<sup>1</sup> In practice, it is rare for compilers to do more than issue a warning. When using either <sup>2</sup> the mpi\_f08 or mpi module, the problem is usually resolved through the assumed-type <sup>3</sup> and assumed-rank declarations of the dummy arguments, or with a compiler-dependent <sup>4</sup> mechanism that overrides type checking for choice arguments.

<sup>5</sup> It is also technically invalid in Fortran to pass a scalar actual argument to an array <sup>6</sup> dummy argument that is not a choice buffer argument. Thus, when using the mpi\_f08 <sup>7</sup> or mpi module, the following code fragment usually generates an error since the dims and <sup>8</sup> periods arguments to MPI\_CART\_CREATE are declared as assumed size arrays INTEGER :: <sup>9</sup> DIMS(\*) and LOGICAL :: PERIODS(\*).

```
10
11
        USE mpi_f08
                          ! or USE mpi
        INTEGER size
12
        CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
13
14
     Although this is a non-conforming MPI call, compiler warnings are not expected (but may
15
     occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit
16
     interfaces.
17
18
     17.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets
19
20
     Arrays with subscript triplets describe Fortran subarrays with or without strides, e.g.,
21
22
         REAL a(100,100,100)
23
         CALL MPI_Send( a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
^{24}
25
     The handling of subscript triplets depends on the value of the constant
26
     MPI_SUBARRAYS_SUPPORTED:
27
         • If MPI_SUBARRAYS_SUPPORTED equals .TRUE .:
28
29
           Choice buffer arguments are declared as TYPE(*), DIMENSION(...). For example,
30
           consider the following code fragment:
31
32
               REAL s(100), r(100)
33
               CALL MPI_Isend(s(1:100:5), 3, MPI_REAL, ..., rq, ierror)
34
               CALL MPI_Wait(rq, status, ierror)
35
               CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL, ..., rq, ierror)
36
               CALL MPI_Wait(rq, status, ierror)
37
38
           In this case, the individual elements s(1), s(6), and s(11) are sent between the start
39
           of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy
40
           s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code
41
           will pass a descriptor to MPI_ISEND that allows MPI to operate directly on s(1), s(6),
42
           s(11), \ldots, s(96). The called MPI_ISEND routine will take only the first three of these
43
           elements due to the type signature "3, MPI_REAL".
44
           All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT,
45
           MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice
46
47
           buffers are copied to a contiguous scratch buffer in the MPI runtime environment.
48
           All datatype descriptions (in the example above, "3, MPI_REAL") read and store
```

data from and to this virtual contiguous scratch buffer. Displacements in MPI derived datatypes are relative to the beginning of this virtual contiguous scratch buffer. 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.

Note that the above definition does not supercede restrictions about buffers used with non-blocking operations (e.g., those specified in Section 3.7.2).

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 release this buffer not before the nonblocking communication has completed (e.g., the MPI\_WAIT routine). Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)

*Rationale.* If MPI\_SUBARRAYS\_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside 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. (*End of rationale.*)

## • 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.

In Fortran, 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.<sup>1</sup>

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:

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

 $^{1}_{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

<sup>&</sup>lt;sup>1</sup>Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

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 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 "simply contiguous" section such as A(1:N) of such an array. ("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.

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 "simply contiguous" section of a contiguous array will also be contiguous.

The same problem can occur with a scalar argument. A compiler may make a copy of 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

```
real :: a
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a
subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end
```

<sup>45</sup> If **a** is copied, MPI\_IRECV will alter the copy when it completes the communication <sup>46</sup> and will not alter **a** itself.

<sup>47</sup> Note that copying will almost certainly occur for an argument that is a non-trivial <sup>48</sup> expression (one with at least one operator or function call), a section that does not

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

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.

If a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this must be employed.

If a compiler makes copies in the calling procedure of arguments that are explicitshape or assumed-size arrays, "simply contiguous" array sections of such arrays, or scalars, and if 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.

### 17.1.13 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

REAL a(100) CALL MPI\_Send( A((/7,9,23,81,82/)), 5, MPI\_REAL, ...)

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.

### 17.1.14 Special Constants

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.

In Fortran, 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 valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, 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.

### 17.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

 $\frac{44}{45}$ 

<sup>1</sup> The following code fragment shows some possible ways to send scalars or arrays of <sup>2</sup> interoperable derived type in Fortran. The example assumes that all data is passed by <sup>3</sup> address.

```
type, BIND(C) :: mytype
5
            integer :: i
6
            real :: x
7
            double precision :: d
8
            logical :: 1
9
         end type mytype
10
11
         type(mytype) :: foo, fooarr(5)
12
         integer :: blocklen(4), type(4)
13
         integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
14
15
         call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
16
         call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
17
         call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
18
         call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
19
20
         base = disp(1)
21
         disp(1) = disp(1) - base
22
         disp(2) = disp(2) - base
23
         disp(3) = disp(3) - base
24
         disp(4) = disp(4) - base
25
26
         blocklen(1) = 1
27
         blocklen(2) = 1
28
         blocklen(3) = 1
29
         blocklen(4) = 1
30
31
         type(1) = MPI_INTEGER
32
         type(2) = MPI_REAL
33
         type(3) = MPI_DOUBLE_PRECISION
34
         type(4) = MPI_LOGICAL
35
36
         call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
37
         call MPI_TYPE_COMMIT(newtype, ierr)
38
39
         call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
40
         ! or
41
         call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
42
         ! expects that base == address(foo%i) == address(foo)
43
44
         call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
45
         call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
46
         extent = disp(2) - disp(1)
47
         1b = 0
48
```

```
call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
call MPI_TYPE_COMMIT(newarrtype, ierr)
```

```
call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
```

Using the derived type variable foo instead of its first basic type element foo%i may be impossible if the MPI library implements choice buffer arguments through overloading instead of using TYPE(\*), DIMENSION(...), or through a non-standardized extension such as !\$PRAGMA IGNORE\_TKR; see Section 17.1.6 on page 614.

To use a derived type in an array requires a correct extent of the datatype handle 10 to take care of the alignment rules applied by the compiler. These alignment rules may 11 imply that there are gaps between the components of a derived type, and also between the 12subsuguent elements of an array of a derived type. The extent of an interoperable derived 13 type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may 14be different because C and Fortran may apply different alignment rules. As recommended 15in the advice to users in Section 4.1.6, one should add an additional fifth structure element 16 with one numerical storage unit at the end of this structure to force in most cases that 17 the array of structures is contiguous. Even with such an additional element, one should 18 keep this resizing due to the special alignment rules that can be used by the compiler for 19structures, as also mentioned in this advice. 20

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER 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. Sending the structure foo should then also be performed by providing it (and not foo%i) as actual argument for MPI\_Send.

## 17.1.16 Optimization Problems, an Overview

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. 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*).

1

2

3 4

5

6

 $\overline{7}$ 

8

9

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 The following compiler optimization strategies (valid for serial code) may cause prob- $\mathbf{2}$ lems in MPI applications: 3 • Code movement and register optimization problems; see Section 17.1.17 on page 636. 4 5• Temporary data movement and temporary memory modifications; see Section 17.1.18 6 on page 643. 7 • Permanent data movement (e.g., through garbage collection); see Section 17.1.19 on 8 9 page 646. 10 Table 17.1 shows the only usage areas where these optimization problems may occur. 11 12Optimization ... ... may cause a problem in 13 following usage areas 141-sided Split Nonbl. Bottom 15Code movement yes yes no ves 16and register optimization 17 Temporary data movement yes yes yes no 18 Permanent data movement yes yes yes yes 19 2021Table 17.1: Occurrence of Fortran optimization problems in several usage areas 2223The solutions in the following sections are based on compromises: 24• to minimize the burden for the application programmer, e.g., as shown in Sections 25"Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on 26pages 638–643, 2728• to minimize the drawbacks on compiler based optimization, and 29 • to minimize the requirements defined in Section 17.1.7 on page 617. 30  $^{31}$ Problems with Code Movement and Register Optimization 32 17.1.17 33 Nonblocking Operations 34 35 If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the 36 compiler will assume that it cannot be modified by a called subroutine unless it is an actual 37 argument of the call. In the most common linkage convention, the subroutine is expected 38 to save and restore certain registers. Thus, the optimizer will assume that a register which 39 held a valid copy of such a variable before the call will still hold a valid copy on return. 40Example 17.1 shows extreme, but allowed, possibilities. MPI\_WAIT on a concurrent 41 thread modifies **buf** between the invocation of MPI\_IRECV and the completion of MPI\_WAIT. 42But the compiler cannot see any possibility that buf can be changed after MPI\_IRECV has 43returned, and may schedule the load of **buf** earlier than typed in the source. The compiler 44has no reason to avoid using a register to hold **buf** across the call to MPI\_WAIT. It also may 45reorder the instructions as illustrated in the rightmost column. 46Due to valid compiler code movement optimizations in Example 17.2, the content of 47buf may already have been 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

<b>Example 17.1</b> Fortran 90	register optimization — extre	ome	1
-	о .		2
Source	compiled as	or compiled as	3
REAL :: buf, b1	REAL :: buf, b1	REAL :: buf, b1	4
<pre>call MPI_IRECV(buf,req)</pre>	<pre>call MPI_IRECV(buf,req)</pre>	<pre>call MPI_IRECV(buf,req)</pre>	5
	register = buf	b1 = buf	6 7
<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>	8
b1 = buf	b1 = register		9
			10
			11
<b>Example 17.2</b> Similar exa	mple with MPI_ISEND		12
Source	compiled as	with a possible MPI-internal	13
		execution sequence	14
REAL :: buf, copy	REAL :: buf, copy	REAL :: buf, copy	15
buf = val	buf = val	buf = val	16
call MPI_ISEND(buf,req)	<pre>call MPI_ISEND(buf,req)</pre>	addr = &buf	17
copy = buf	copy= buf	copy = buf	18
<pre>call MPI_WAIT(req,)</pre>	<pre>buf = val_overwrite call MPI_WAIT(req,)</pre>	<pre>buf = val_overwrite call send(*addr) ! within</pre>	19
call mi_wall(leq,)	Call MILWAII(Ieq,)	· MPI_WAIT	20
<pre>buf = val_overwrite</pre>		· · · · <b></b> · · ·	21
			22
			23
buf in $MPI_WAIT$ (or in a set	econd thread between the sta	rt of MPI_ISEND and the end of	24
MPI_WAIT).			25
Such register optimizati	on is based on moving code; h	nere, the access to buf was moved	26
from after $MPI_WAIT$ to before	ore MPI_WAIT. Note that code	e movement may also occur across	27
subroutine boundaries when	subroutines or functions are	inlined.	28
This register optimization/code movement problem for nonblocking operations does			29
-	7	ations, because in theBEGIN	30
	-	an actual argument. The register	31
optimization / code movement problem for $MPI\_BOTTOM$ and derived $MPI$ datatypes may			
occur in each blocking and nonblocking communication call, as well as in each parallel file			

### **One-sided** Communication

I/O operation.

An example with instruction reordering due to register optimization can be found in Section 11.7.4 on page 463.

### MPI\_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI\_SEND, MPI\_RECV, etc., that hides the actual variables involved in the communication. 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 referenced in the call. Also attention must be paid if MPI

42

```
1
     operations are used that run in parallel with the user's application.
\mathbf{2}
         Example 17.3 shows what Fortran compilers are allowed to do.
3
4
     Example 17.3 Fortran 90 register optimization.
5
     This source ...
                                                   can be compiled as:
6
7
     call MPI_GET_ADDRESS(buf,bufaddr,
                                                  call MPI_GET_ADDRESS(buf,...)
8
                      ierror)
9
     call MPI_TYPE_CREATE_STRUCT(1,1,
                                                  call MPI_TYPE_CREATE_STRUCT(...)
10
                      bufaddr,
11
                      MPI_REAL,type,ierror)
12
     call MPI_TYPE_COMMIT(type,ierror)
                                                  call MPI_TYPE_COMMIT(...)
13
     val_old = buf
                                                  register = buf
14
                                                  val_old = register
15
     call MPI_RECV(MPI_BOTTOM,1,type,...)
                                                  call MPI_RECV(MPI_BOTTOM,...)
16
                                                  val_new = register
     val_new = buf
17
18
19
         In Example 17.3, the compiler does not invalidate the register because it cannot see
20
     that MPI_RECV changes the value of buf. The access to buf is hidden by the use of
21
     MPI_GET_ADDRESS and MPI_BOTTOM.
22
23
     Example 17.4 Similar example with MPI_SEND
24
     This source ...
                                                   can be compiled as:
25
26
     ! buf contains val_old
                                                  ! buf contains val_old
27
     buf = val_new
28
     call MPI_SEND(MPI_BOTTOM,1,type,...)
                                                  call MPI_SEND(...)
29
     ! with buf as a displacement in type
                                                  ! i.e. val_old is sent
30
                                                  !
^{31}
                                                   ! buf=val_new is moved to here
32
                                                  ! and detected as dead code
33
                                                   ! and therefore removed
34
                                                   i
35
     buf = val_overwrite
                                                  buf = val_overwrite
36
37
38
         In Example 17.4, several successive assignments to the same variable buf can be com-
39
```

<sup>39</sup> bined in a way such that only the last assignment is executed. "Successive" means that <sup>40</sup> no interfering load access to this variable occurs between the assignments. The compiler <sup>41</sup> cannot detect that the call to MPI\_SEND statement is interfering because the load access <sup>42</sup> to buf is hidden by the usage of MPI\_BOTTOM.

<sup>44</sup> Solutions

43

45

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations with the send and receive buffers used in nonblocking operations, or in operations in which MPI\_BOTTOM is used, or if datatype handles that combine several variables are used:

- Use of the Fortran ASYNCHRONOUS attribute.
- Use of the helper routine MPI\_F\_SYNC\_REG, or an equivalent user-written dummy routine.
- Declare the buffer as a Fortran module variable or within a Fortran common block.
- Use of the Fortran VOLATILE attribute.

Example 17.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)
                               ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)
#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
 CALL MPI_Waitall(4,req,...)
 DO i=1,100 ! compute all new local data
   bnew(i) = function(b(i-1), b(i), b(i+1))
 END DO
#endif
#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
 DO i=2,99 ! compute only elements for which halo data is not needed
   bnew(i) = function(b(i-1), b(i), b(i+1))
 END DO
 CALL MPI_Waitall(4, req,...)
 i=1 ! compute leftmost element
   bnew(i) = function(b(i-1), b(i), b(i+1))
 i=100 ! compute rightmost element
   bnew(i) = function(b(i-1), b(i), b(i+1))
#endif
```

Each of these methods solves the problems of code movement and register optimization, but may incur various degrees of performance impact, and may not be usable in every

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 application context. These methods may not be guaranteed by the Fortran standard, but  $\mathbf{2}$ they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated 3 compiler suite according to the requirements listed in Section 17.1.7 on page 617. The 4 performance impact of using MPI\_F\_SYNC\_REG is expected to be low, that of using module  $\mathbf{5}$ variables or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using 6 the VOLATILE attribute is expected to be high or very high. Note that there is one attribute  $\overline{7}$ that cannot be used for this purpose: the Fortran TARGET attribute does not solve code 8 movement problems in MPI applications.

9 10

 $^{31}$ 

32

33

34

35

36

37

38

## The Fortran ASYNCHRONOUS Attribute

11Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping 12unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed 13 while the buffer is affected by a pending asynchronous Fortran input/output operation (since 14Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the 15extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the 16Fortran compiler implements asynchronous Fortran input/output operations with blocking 17I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through 18 code movements across routine calls, and the buffer itself from temporary and permanent 19 data movements. If the choice buffer dummy argument of a nonblocking MPI routine is 20declared with ASYNCHRONOUS (which is mandatory for the mpi\_f08 module, with allowable 21exceptions listed in Section 17.1.6 on page 614), then the compiler has to guarantee call by 22reference and should report a compile-time error if call by reference is impossible, e.g., if 23vector subscripts are used. The MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if  $^{24}$ both the protection of the actual buffer argument through ASYNCHRONOUS according to the 25TS 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the 26Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to 27.FALSE.. 28

The ASYNCHRONOUS attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

"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 status changed."

In Example 17.5 Case (a) on page 639, 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 has 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) is pending. This is a contradiction to the rule that *for input communication*, *a pending communication affector shall not be referenced*. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjoint subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 17.9 on page 647.

If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the ASYNCHRONOUS attribute.

The problems with MPI\_BOTTOM, as shown in Example 17.3 and Example 17.4, can also be solved by declaring the buffer **buf** with the ASYNCHRONOUS attribute.

In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS.

Calling MPI\_F\_SYNC\_REG

The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. The MPI library provides the MPI\_F\_SYNC\_REG routine for this purpose; see Section 17.1.8 on page 619.

• The problems illustrated by the Examples 17.1 and 17.2 can be solved by calling MPI\_F\_SYNC\_REG(buf) once immediately after MPI\_WAIT.

Example 17.1	Example 17.2	29
can be solved with	can be solved with	30
call MPI_IRECV(buf,req)	buf = val	31
	call MPI_ISEND(buf,req)	32
	copy = buf	33
<pre>call MPI_WAIT(req,)</pre>	call MPI_WAIT(req,)	34 35
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)	36
b1 = buf	<pre>buf = val_overwrite</pre>	37

The call to MPI\_F\_SYNC\_REG(buf) prevents moving the last line before the MPI\_WAIT call. Further calls to MPI\_F\_SYNC\_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI\_WAIT and before buf=val\_overwrite.

• The problems illustrated by the Examples 17.3 and 17.4 can be solved with two additional MPI\_F\_SYNC\_REG(buf) statements; one directly before MPI\_RECV/ MPI\_SEND, and one directly after this communication operation.

Example 17.3	Example 17.4	47
can be solved with	can be solved with	48

 $\mathbf{2}$ 

 $^{24}$ 

 $45 \\ 46$ 

1 2 3	call MPI_F_SYNC_REG(buf) call MPI_RECV(MPI_BOTTOM, call MPI_F_SYNC_REG(buf)	.)	<pre>call MPI_F_SYNC_REG(buf) call MPI_SEND(MPI_BOTTOM,) call MPI_F_SYNC_REG(buf)</pre>	
4 5 6 7	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 any subsequent access to buf is not moved before MPI_RECV/SEND.			
8 9 10 11 12 13 14	• In the example in Section 11.7.4 on page 463, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 17.1, 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			
15 16 17	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.			
18 19	Source of Process 1	Source o	f Process 2	
20	bbbb = 777	buff = 99		
21		call MPI	_F_SYNC_REG(buff)	
22	call MPI_WIN_FENCE	call MPI	_WIN_FENCE	
23	call MPI_PUT(bbbb			
24	into buff of process 2)			
25				
26	call MPI_WIN_FENCE		_WIN_FENCE	
27	call MPI_F_SYNC_REG(bbbb)		_F_SYNC_REG(buff)	
28 29		ccc = bu:	I I	
30 31 32	• The temporary memory modification problem, i.e., Example 17.6 on page 644, can <b>not</b> be solved with this method.			
33 34	A User Defined Routine Instead of MPI_F	_SYNC_REG		
35 36	Instead of MPI_F_SYNC_REG, one can also use a user defined external subroutine, which is separately compiled:			
37	subroutine DD(buf)			
38 39	integer buf			
40	end			
41				
42		-	interface for the external subroutine,	
43	it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler			
44	does not know this and has to assume the			
45	to MPI_RECV with MPI_BOTTOM as but	ner might be	replaced by	
46	call DD(buf)			
47	call MPI_RECV(MPI_BOTTOM,	)		
48	call DD(buf)			

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 MPI\_F\_SYNC\_REG or one of the other possibilities. 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 in accordance with the rules specified in Section 17.1.7 on page 617.

# Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a 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. The compiler will then have to assume that the MPI procedure may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block.

- 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.
- 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.

# The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution of MPI programs is discouraged.

# The Fortran TARGET Attribute

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 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.

*Rationale.* The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

# 17.1.18 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 17.5,

```
1
      Example 17.6 Overlapping Communication and Computation.
\mathbf{2}
3
     USE mpi_f08
4
     REAL :: buf(100,100)
\mathbf{5}
      CALL MPI_Irecv(buf(1,1:100),...req,...)
6
     DO j=1,100
\overline{7}
        DO i=2,100
8
          buf(i,j)=....
9
        END DO
10
     END DO
11
      CALL MPI_Wait(req,...)
12
13
14
      Example 17.7 The compiler may substitute the nested loops through loop fusion.
15
16
     REAL :: buf(100,100), buf_1dim(10000)
17
      EQUIVALENCE (buf(1,1), buf_1dim(1))
18
      CALL MPI_Irecv(buf(1,1:100),...req,...)
19
     tmp(1:100) = buf(1,1:100)
20
      DO j=1,10000
21
        buf_1dim(h)=...
22
     END DO
     buf(1,1:100) = tmp(1:100)
23
^{24}
     CALL MPI_Wait(req,...)
25
26
27
      Case (b) on page 639. Example 17.6 on page 644 also shows a possibility that could be
28
      problematic.
29
          In the compiler-generated, possible optimization in Example 17.7, buf(100,100) from
30
      Example 17.6 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblock-
31
      ing receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused
32
      loop is temporarily using this part of the buffer. When the tmp data is written back to buf,
33
      the previous data of buf(1,1:100) is restored and the received data is lost. The principle
34
      behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved
35
      to tmp.
36
          Example 17.8 shows a second possible optimization. The whole array is temporarily
37
      moved to local_buf.
38
          When storing local_buf back to the original location buf, then this implies overwriting
39
      the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this
40
     storing back of local_buf is therefore likely to interfere with asynchronously received data
^{41}
     in buf(1,1:100).
42
          Note that this problem may also occur:
43
         • With the local buffer at the origin process, between an RMA communication call and
44
           the ensuing synchronization call; see Chapter 11 on page 401.
45
46
         • With the window buffer at the target process between two ensuing RMA synchroniza-
47
           tion calls.
48
```

**Example 17.8** Another optimization is based on the usage of a separate memory storage area, e.g., in a GPU.

```
REAL :: buf(100,100), local_buf(100,100)
CALL MPI_Irecv(buf(1,1:100),...req,...)
local_buf = buf
DO j=1,100
    DO i=2,100
    local_buf(i,j)=....
END DO
END DO
buf = local_buf ! may overwrite asynchronously received
        ! data in buf(1,1:100)
CALL MPI_Wait(req,...)
```

• With the local buffer in MPI parallel file I/O split collective operations between the ...\_BEGIN and ...\_END calls; see Section 13.5.4 on page 523.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 640 of Section 17.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 17.9 on page 647 and in Example 17.10 on page 648.

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 code fragments shown in Example 17.6 and 17.7.

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 any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the ASYNCHRONOUS attribute. In 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 operations.

Rationale. This is a strong restriction for application programs. To weaken this <sup>46</sup> restriction, a new or modified asynchronous feature in the Fortran language would <sup>47</sup> be necessary: an asynchronous attribute that can be used on parts of an array and <sup>48</sup>

together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 17.9 on page 647 (which is a solution for the problem shown in Example 17.5 5on page 639 and in Example 17.10 on page 648 (which is a solution for the problem shown 6 in Example 17.8 on page 645), the array is split into inner and halo part and both disjoint 7 parts are passed to a subroutine **separated\_sections**. This routine overlaps the receiving 8 of the halo data and the calculations on the inner part of the array. In a second step, the 9 whole array is used to do the calculation on the elements where inner+halo is needed. Note 10 that the halo and the inner area are strided arrays. Those can be used in non-blocking 11 communication only with a TS 29113 based MPI library. 12

 $13 \\ 14$ 

15

16

17

18

19 20

21

22

23

 $^{24}$ 

1

2

3

4

# 17.1.19 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.An implementation with automatic garbage collection 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 17.1.7 on page 617.

30 31

# 17.1.20 Comparison with C

32 In C, subroutines which modify variables that are not in the argument list will not cause 33 register optimization problems. This is because taking pointers to storage objects by using 34the & operator and later referencing the objects by indirection on the pointer is an integral 35 part of the language. A C compiler understands the implications, so that the problem should 36 not occur, in general. However, some compilers do offer optional aggressive optimization 37 levels which may not be safe. Problems due to temporary memory modifications can also 38occur in C. As above, the best advice is to avoid the problem: use different variables for 39 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 40operation is pending.

- 41
- 42
- 43
- 44 45
- 40
- 46
- 47 48

**Example 17.9** Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
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))
i=1 ! compute leftmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
  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)
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
                               ..., right, ..., req(4), ...)
CALL MPI_Isend(b_inner(100),
                                                                                 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 sent at the same time.
                                                                                 38
  ! This is allowed based on the rules for ASYNCHRONOUS.
                                                                                 39
END DO
CALL MPI_Waitall(4,req,...)
END SUBROUTINE
                                                                                 42
```

8

9 10

11

15

16

17

18

26

40

41

```
1
\mathbf{2}
3
4
5
6
7
8
9
10
11
12
13
14
     Example 17.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
15
     USE mpi_f08
16
     REAL :: buf(100,100)
17
     CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
18
     END
19
20
     SUBROUTINE separated_sections(buf_halo, buf_inner)
21
     REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
22
     REAL :: buf_inner(2:100,1:100)
23
     REAL :: local_buf(2:100,100)
^{24}
25
     CALL MPI_Irecv(buf_halo(1,1:100),...req,...)
26
     local_buf = buf_inner
27
     DO j=1,100
28
       DO i=2,100
29
          local_buf(i,j)=....
30
       END DO
31
     END DO
32
     buf_inner = local_buf ! buf_halo is not touched!!!
33
34
     CALL MPI_Wait(req,...)
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

# 17.2 Language Interoperability

## 17.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) 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.

**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.

## 17.2.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERs, can be passed to a C program. We also assume that Fortran and C have addresssized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI\_OFFSET\_KIND) can be passed from Fortran to C as MPI\_Offset.

## 17.2.3 Initialization

A call to MPI\_INIT or MPI\_INIT\_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI\_INIT in order to propagate values for argc and argv to all

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

 $46 \\ 47$ 

2 result in a loss of this ability. (End of advice to users.) 3 The function MPI\_INITIALIZED returns the same answer in all languages. 4 The function MPI\_FINALIZE finalizes the MPI environments for all languages. 5The function MPI\_FINALIZED returns the same answer in all languages. 6 The function MPI\_ABORT kills processes, irrespective of the language used by the 7 caller or by the processes killed. 8 9 The MPI environment is initialized in the same manner for all languages by 10 MPI\_INIT. E.g., MPI\_COMM\_WORLD carries the same information regardless of language: 11same processes, same environmental attributes, same error handlers. Information can be added to info objects in one language and retrieved in another. 1213 Advice to users. The use of several languages in one MPI program may require the 14use of special options at compile and/or link time. (End of advice to users.) 1516Advice to implementors. Implementations may selectively link language specific MPI 17 libraries only to codes that need them, so as not to increase the size of binaries for codes 18 that use only one language. The MPI initialization code need perform initialization for 19 a language only if that language library is loaded. (End of advice to implementors.) 202117.2.4 Transfer of Handles 22 23Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran  $^{24}$ handles to C handles. There is no direct access to C handles in Fortran. 25The type definition MPI\_Fint is provided in C for an integer of the size that matches a 26Fortran INTEGER; usually, MPI\_Fint will be equivalent to int. With the Fortran mpi module 27or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in 28the following conversion functions. With the Fortran mpi\_f08 module, a Fortran handle is a 29BIND(C) derived type that contains an INTEGER component named MPI\_VAL. This INTEGER 30 value can be used in the following conversion functions.  $^{31}$ The following functions are provided in C to convert from a Fortran communicator han-32 dle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.4 33 on page 20. 34MPI\_Comm MPI\_Comm\_f2c(MPI\_Fint comm) 35 If comm is a valid Fortran handle to a communicator, then MPI\_Comm\_f2c returns a 36 valid C handle to that same communicator; if  $comm = MPI_COMM_NULL$  (Fortran value), 37 then MPI\_Comm\_f2c returns a null C handle; if comm is an invalid Fortran handle, then 38 39 MPI\_Comm\_f2c returns an invalid C handle. MPI\_Fint MPI\_Comm\_c2f(MPI\_Comm comm) 4041 The function MPI\_Comm\_c2f translates a C communicator handle into a Fortran handle 42to the same communicator; it maps a null handle into a null handle and an invalid handle 43 into an invalid handle. 44 Similar functions are provided for the other types of opaque objects. 45MPI\_Datatype MPI\_Type\_f2c(MPI\_Fint datatype) 4647MPI\_Fint MPI\_Type\_c2f(MPI\_Datatype datatype)

executing processes. Use of the Fortran version of MPI\_INIT to initialize MPI may

1

	1
MPI_Group MPI_Group_f2c(MPI_Fint group)	1 2
MPI_Fint MPI_Group_c2f(MPI_Group group)	3
MPI_Request MPI_Request_f2c(MPI_Fint request)	4
MPI_Fint MPI_Request_c2f(MPI_Request request)	5 6
MPI_File MPI_File_f2c(MPI_Fint file)	7
MPI_Fint MPI_File_c2f(MPI_File file)	8 9
MPI_Win MPI_Win_f2c(MPI_Fint win)	10
MPI_Fint MPI_Win_c2f(MPI_Win win)	11 12
MPI_Op MPI_Op_f2c(MPI_Fint op)	13
MPI_Fint MPI_Op_c2f(MPI_Op op)	14 15
MPI_Info MPI_Info_f2c(MPI_Fint info)	16
	17
MPI_Fint MPI_Info_c2f(MPI_Info info)	18 19
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	20
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	21
MPI_Message MPI_Message_f2c(MPI_Fint message)	22 23
MPI_Fint MPI_Message_c2f(MPI_Message message)	24
······································	25
<b>Example 17.11</b> The example below illustrates how the Fortran MPI function	26
MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function	27 28
MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C	28 29
interface is assumed where a Fortran function is all upper case when referred to from C and	30
arguments are passed by addresses.	31
! FORTRAN PROCEDURE	32
SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)	33
INTEGER :: DATATYPE, IERR	34 35
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	36
RETURN END	37
	38
/* C wrapper */	39
word MDI V TYDE COMMIT( MDI Fint of bondle MDI Fint views)	40 41
<pre>void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr) {</pre>	42
MPI_Datatype datatype;	43
	44
<pre>datatype = MPI_Type_f2c( *f_handle);</pre>	45
<pre>*ierr = (MPI_Fint)MPI_Type_commit( &amp;datatype);</pre>	46 47
<pre>*f_handle = MPI_Type_c2f(datatype); return:</pre>	47
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.*)

17.2.5 Status

The following two procedures are provided in C to convert from a Fortran (with the mpi module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

int MPI\_Status\_f2c(const MPI\_Fint \*f\_status, MPI\_Status \*c\_status)

If f\_status is a valid Fortran status, but not the Fortran value of MPI\_STATUS\_IGNORE 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 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 exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Figure 17.1 on page 653. (*End of advice to users.*)

}

1

2 3

4

5 6

7

8

9

10

11 12

13

14

15

16

17 18

19

25

26

27

28

29

30

 $^{31}$ 

32

33

40

41

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 17.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.

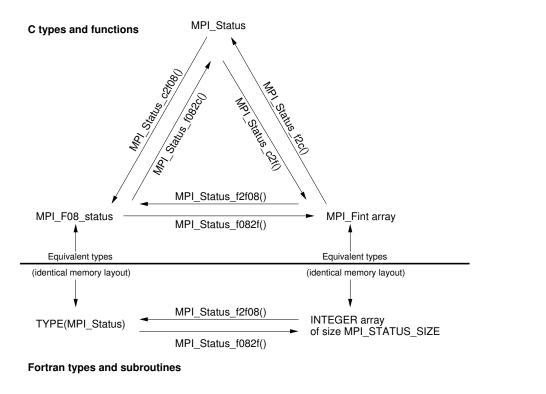


Figure 17.1: Status conversion routines

## 

This C routine converts a Fortran mpi\_f08 TYPE(MPI\_Status) into a C MPI\_Status. int MPI\_Status\_c2f08(const MPI\_Status \*c\_status, MPI\_F08\_status \*f08\_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.

```
1
         Conversion between the two Fortran versions of a status can be done with:
\mathbf{2}
3
     MPI_STATUS_F2F08(f_status, f08_status)
4
5
       IN
                 f_status
                                            status object declared as array
6
       OUT
                 f08_status
                                            status object declared as named type
7
8
     int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
9
10
     MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C)
11
          INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
12
         TYPE(MPI_Status), INTENT(OUT) :: f08_status
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
15
          INTEGER :: F_STATUS(MPI_STATUS_SIZE)
16
         TYPE(MPI_Status) :: F08_STATUS
17
         INTEGER IERROR
18
19
         This routine converts a Fortran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array
20
     into a Fortran mpi_f08 TYPE(MPI_Status).
21
22
     MPI_STATUS_F082F(f08_status, f_status)
23
24
       IN
                 f08_status
                                            status object declared as named type
25
       OUT
                f_status
                                            status object declared as array
26
27
     int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
28
29
     MPI_Status_f082f(f08_status, f_status, ierror) BIND(C)
30
         TYPE(MPI_Status), INTENT(IN) :: f08_status
31
          INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
         TYPE(MPI_Status) :: F08_STATUS
35
         INTEGER :: F_STATUS(MPI_STATUS_SIZE)
36
         INTEGER IERROR
37
38
         This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER,
39
     DIMENSION(MPI_STATUS_SIZE) status array.
40
41
            MPI Opaque Objects
     17.2.6
42
43
     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
44
```

45in the previous section can be used to pass references to MPI objects from language to 46language. An object created in one language can be accessed, modified or freed in another 47language. 48

654

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 Section 17.2.9 on page 661).

#### Example 17.12

```
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
! create an absolute datatype for array R
                                                                                    20
AOBLEN(1) = 5
                                                                                   21
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
                                                                                   22
AOTYPE(1) = MPI_REAL
                                                                                   23
                                                                                    24
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
                                                                                    25
CALL C_ROUTINE(TYPE)
/* C code */
                                                                                    27
                                                                                    28
void C_ROUTINE(MPI_Fint *ftype)
                                                                                    29
{
                                                                                    30
   int count = 5;
   int lens[2] = \{1, 1\};
   MPI_Aint displs[2];
                                                                                    33
   MPI_Datatype types[2], newtype;
                                                                                   34
                                                                                   35
   /* create an absolute datatype for buffer that consists
                                                                 */
                                                                                   36
   /* of count, followed by R(5)
                                                                 */
                                                                                   37
                                                                                    38
   MPI_Get_address(&count, &displs[0]);
                                                                                    39
   displs[1] = 0;
   types[0] = MPI_INT;
                                                                                    41
   types[1] = MPI_Type_f2c(*ftype);
                                                                                    42
   MPI_Type_create_struct(2, lens, displs, types, &newtype);
                                                                                    43
   MPI_Type_commit(&newtype);
                                                                                    44
   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   /* the message sent contains an int count of 5, followed
                                                                 */
   /* by the 5 REAL entries of the Fortran array R.
                                                                 */
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

1213

14

1516

17

18

19

26

31

32

40

45

46

47

Advice to implementors. The following implementation can be used: MPI addresses, as returned by MPI\_GET\_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI\_BOTTOM. Thus, if MPI\_BOTTOM is zero then a send or receive call with buf = MPI\_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI\_BOTTOM is not zero, then the implementation has to be slightly different. A test is performed to check whether buf = MPI\_BOTTOM. If true, then the base address is zero, otherwise it is buf. In particular, if MPI\_BOTTOM does not have the same value in Fortran and C, then an additional test for buf = MPI\_BOTTOM is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI\_BOTTOM even in C, so as to distinguish it from a NULL pointer. If MPI\_BOTTOM = c then one can still avoid the test buf = MPI\_BOTTOM, by using the displacement from MPI\_BOTTOM, 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.*)

23 Callback Functions

<sup>24</sup> MPI calls may associate callback functions with MPI objects: error handlers are associ-<sup>25</sup> ated with communicators and files, attribute copy and delete functions are associated with <sup>26</sup> attribute keys, reduce operations are associated with operation objects, etc. In a multilan-<sup>27</sup> guage environment, a function passed in an MPI call in one language may be invoked by an <sup>28</sup> MPI call in another language. MPI implementations must make sure that such invocation <sup>29</sup> 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 270. (*End of advice to users.*)

# Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length 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**

All predefined named and unnamed datatypes as listed in Section 5.9.2 on page 176 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

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 and Fortran datatypes. (*End of advice to users.*)

#### 17.2.7 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.

Advice to implementors. This requires that attributes be tagged either as "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.*)

The attribute manipulation functions described in Section 6.7 on page 265 defines 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 pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI\_ATTR\_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI\_ATTR\_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 6.7 on page 265. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer-valued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from 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 stored with Fortran MPI\_XXX\_SET\_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI\_ATTR\_PUT. When an address-valued attribute is accessed from 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 are used, and an 

 $\overline{7}$ 

 $^{31}$ 

```
1
     integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if
\mathbf{2}
     deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and
3
     MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr.
4
     Example 17.13
5
         A. Setting an attribute value in C
6
7
     int set_val = 3;
8
     struct foo set_struct;
9
10
     /* Set a value that is a pointer to an int */
11
12
     MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
13
     /* Set a value that is a pointer to a struct */
14
     MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
15
     /* Set an integer value */
16
     MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
17
18
         B. Reading the attribute value in C
19
20
     int flag, *get_val;
21
     struct foo *get_struct;
22
23
     /* Upon successful return, get_val == &set_val
^{24}
         (and therefore *get_val == 3) */
25
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
26
     /* Upon successful return, get_struct == &set_struct */
27
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
28
     /* Upon successful return, get_val == (void*) 17 */
                i.e., (MPI_Aint) get_val == 17 */
^{29}
     /*
30
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
31
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
32
33
     LOGICAL FLAG
34
     INTEGER IERR, GET_VAL, GET_STRUCT
35
36
     ! Upon successful return, GET_VAL == &set_val, possibly truncated
37
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
38
     ! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
39
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
40
     ! Upon successful return, GET_VAL == 17
41
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
42
43
         D. Reading the attribute value with Fortran MPI-2 calls
44
45
46
47
48
```

```
1
LOGICAL FLAG
                                                                                       \mathbf{2}
INTEGER IERR
                                                                                       3
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
                                                                                       4
! Upon successful return, GET_VAL == &set_val
                                                                                       5
                                                                                       6
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
                                                                                       7
! Upon successful return, GET_STRUCT == &set_struct
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
                                                                                       8
! Upon successful return, GET_VAL == 17
                                                                                       9
                                                                                      10
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                       11
                                                                                       12
Example 17.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
                                                                                      13
                                                                                      14
INTEGER IERR, VAL
                                                                                       15
VAL = 7
                                                                                       16
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
                                                                                       17
                                                                                       18
    B. Reading the attribute value in C
                                                                                       19
                                                                                      20
int flag;
                                                                                      21
int *value;
                                                                                      22
                                                                                      23
/* Upon successful return, value points to internal MPI storage and
                                                                                       ^{24}
   *value == (int) 7 */
                                                                                      25
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
                                                                                       26
                                                                                      27
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                      28
                                                                                      29
LOGICAL FLAG
                                                                                       30
INTEGER IERR, VALUE
                                                                                       31
                                                                                       32
! Upon successful return, VALUE == 7
                                                                                      33
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      34
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      35
                                                                                      36
LOGICAL FLAG
                                                                                      37
INTEGER IERR
                                                                                       38
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
                                                                                       39
                                                                                       40
! Upon successful return, VALUE == 7 (sign extended)
                                                                                      41
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      42
                                                                                      43
                                                                                      44
Example 17.15 A. Setting an attribute value via a Fortran MPI-2 call
                                                                                       45
                                                                                       46
                                                                                       47
```

```
1
     INTEGER IERR
\mathbf{2}
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
3
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
4
     VALUE1 = 42
\mathbf{5}
     VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
6
7
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
8
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
9
10
         B. Reading the attribute value in C
11
     int flag;
12
     MPI_Aint *value1, *value2;
13
14
     /* Upon successful return, value1 points to internal MPI storage and
15
        *value1 == 42 */
16
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
17
     /* Upon successful return, value2 points to internal MPI storage and
18
        *value2 == 2^40 */
19
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
20
21
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
22
23
     LOGICAL FLAG
24
     INTEGER IERR, VALUE1, VALUE2
25
26
     ! Upon successful return, VALUE1 == 42
27
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
28
     ! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
29
     ! needed (i.e., the least significant part of the attribute word)
30
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
^{31}
32
         D. Reading the attribute value with Fortran MPI-2 calls
33
34
     LOGICAL FLAG
35
     INTEGER IERR
36
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
37
38
     ! Upon successful return, VALUE1 == 42
39
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
40
     ! Upon successful return, VALUE2 == 2^40
41
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
42
43
         The predefined MPI attributes can be integer valued or address-valued. Predefined
44
     integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to
45
     the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran,
46
     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.

Address-valued predefined attributes, such as MPI\_WIN\_BASE behave as if they were put by a C call, i.e., in Fortran, MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI\_Win\_get\_attr(win, MPI\_WIN\_BASE, &p, &flag) will return in p a pointer to the window base, cast to (void \*).

*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.)

# 17.2.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

# 17.2.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI\_INT, MPI\_COMM\_WORLD, MPI\_ERRORS\_RETURN, MPI\_SUM, etc.) These handles need to be converted, as explained in Section 17.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI\_MAX\_OBJECT\_NAME];

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

2

3

4

5 6

7

8

9

10

11

12

13

### (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 in Fortran must be 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. See the advice to implementors in the Datatypes subsection in Section 17.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (End of rationale.)

#### 14 15 16

17

# 17.2.10 Interlanguage Communication

The type matching rules for communication 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 17.16 In the example below, a Fortran array is sent from Fortran and received in C.

```
27
     ! FORTRAN CODE
28
     SUBROUTINE MYEXAMPLE()
29
     USE mpi_f08
30
     REAL :: R(5)
^{31}
     INTEGER :: IERR, MYRANK, AOBLEN(1)
32
     TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
33
     INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
34
35
     ! create an absolute datatype for array \ensuremath{\mathtt{R}}
36
     AOBLEN(1) = 5
37
     CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
38
     AOTYPE(1) = MPI_REAL
39
     CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
40
     CALL MPI_TYPE_COMMIT(TYPE, IERR)
41
42
     CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
43
     IF (MYRANK.EQ.O) THEN
        CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
44
45
     ELSE
46
        CALL C_ROUTINE(TYPE%MPI_VAL)
47
     END IF
48
     END SUBROUTINE
```

```
/* C code */
void C_ROUTINE(MPI_Fint *fhandle)
{
    MPI_Datatype type;
    MPI_Status status;
    type = MPI_Type_f2c(*fhandle);
    MPI_Recv( MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI\_INTEGER and be received with datatype MPI\_INT. However, such code is not portable.

# Annex A

# Language Bindings Summary

In this section we summarize the specific bindings for C and Fortran. 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 names are listed below. Constants with the type const int may also be implemented as literal integer constants substituted by the preprocessor.

	24
Error classes	25
C type: const int (or unnamed enum)	26
Fortran type: INTEGER	27
MPI_SUCCESS	28
MPI_ERR_BUFFER	29
MPI_ERR_COUNT	30
MPI_ERR_TYPE	31
MPI_ERR_TAG	32
MPI_ERR_COMM	33
MPI_ERR_RANK	34
MPI_ERR_REQUEST	35
MPI_ERR_ROOT	36
MPI_ERR_GROUP	37
MPI_ERR_OP	38
MPI_ERR_TOPOLOGY	39
MPI_ERR_DIMS	40
MPI_ERR_ARG	41
MPI_ERR_UNKNOWN	42
MPI_ERR_TRUNCATE	43
MPI_ERR_OTHER	44
MPI_ERR_INTERN	45
MPI_ERR_PENDING	46
(Continued on next page)	47
	48

Error classes (continued)
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_ERR_IN_STATUS
MPI_ERR_ACCESS
MPI_ERR_AMODE
MPI_ERR_ASSERT
MPI_ERR_BAD_FILE
MPI_ERR_BASE
MPI_ERR_CONVERSION
MPI_ERR_DISP
MPI_ERR_DUP_DATAREP
MPI_ERR_FILE_EXISTS
MPI_ERR_FILE_IN_USE
MPI_ERR_FILE
MPI_ERR_INFO_KEY
MPI_ERR_INFO_NOKEY
MPI_ERR_INFO_VALUE
MPI_ERR_INFO
MPI_ERR_IO
MPI_ERR_KEYVAL
MPI_ERR_LOCKTYPE
MPI_ERR_NAME
MPI_ERR_NO_MEM
MPI_ERR_NOT_SAME
MPI_ERR_NO_SPACE
MPI_ERR_NO_SUCH_FILE
MPI_ERR_PORT
MPI_ERR_QUOTA
MPI_ERR_READ_ONLY
MPI_ERR_RMA_ATTACH
MPI_ERR_RMA_CONFLICT
MPI_ERR_RMA_RANGE
MPI_ERR_RMA_SHARED
MPI_ERR_RMA_SYNC
MPI_ERR_RMA_FLAVOR
MPI_ERR_SERVICE
MPI_ERR_SIZE
MPI_ERR_SPAWN
MPI_ERR_UNSUPPORTED_DATAREP
MPI_ERR_UNSUPPORTED_DATAREP
MPI_ERR_WIN
(Continued on next page)

- 47
- $^{48}$

Error classes (continued)	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
MPI_T_ERR_CANNOT_INIT	4
MPI_T_ERR_NOT_INITIALIZED	5
MPI_T_ERR_MEMORY	6
MPI_T_ERR_INVALID_INDEX	7
MPI_T_ERR_INVALID_ITEM	8
MPI_T_ERR_INVALID_SESSION	9
MPI_T_ERR_INVALID_HANDLE	10
MPI_T_ERR_OUT_OF_HANDLES	11
MPI_T_ERR_OUT_OF_SESSIONS	12
MPI_T_ERR_CVAR_SET_NOT_NOW	13
MPI_T_ERR_CVAR_SET_NEVER	14
MPI_T_ERR_PVAR_NO_WRITE	15
MPI_T_ERR_PVAR_NO_STARTSTOP	16
MPI_T_ERR_PVAR_NO_ATOMIC	17
MPI_ERR_LASTCODE	18
	19
Buffer Address Constants	20
C type: void * const	21
Fortran type: (predefined memory location) <sup><math>1</math></sup>	22
MPI_BOTTOM	23
MPI_IN_PLACE	24
$^{-1}$ Note that in Fortran these constants are not usable for initialization	 1 25
expressions or assignment. See Section $2.5.4$ on page 15.	26
	27
Assorted Constants	28
C type: const int (or unnamed enum)	29
Fortran type: INTEGER	30
MPI_PROC_NULL	31
MPI_ANY_SOURCE	32
MPI_ANY_TAG	33
MPI_UNDEFINED	34
MPI_BSEND_OVERHEAD	35
MPI_KEYVAL_INVALID	36
MPI_LOCK_EXCLUSIVE	37
MPI_LOCK_SHARED	38
MPI_ROOT	39
No Process Message Handle	40
C type: MPI_Message	41
Fortran type: INTEGER or TYPE(MPI_Message)	42
MPI_MESSAGE_NO_PROC	43
	44
Fortran Support Method Specific Constants	45
Fortran type: LOGICAL	46
MPI_SUBARRAYS_SUPPORTED (Fortran only)	47
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)	48

1	Status size and reserved index values (Fortran only)
2	Fortran type: INTEGER
3	MPI_STATUS_SIZE
4	MPI_SOURCE
5	MPI_TAG
6	MPI_ERROR
7	
8	Variable Address Size (Fortran only)
9	Fortran type: INTEGER
10	MPI_ADDRESS_KIND
11	MPI_COUNT_KIND
12	MPI_INTEGER_KIND
13	MPI_OFFSET_KIND
14	
15	Error-handling specifiers
16	C type: MPI_Errhandler
17	Fortran type: INTEGER or TYPE(MPI_Errhandler)
18	MPI_ERRORS_ARE_FATAL
19	MPI_ERRORS_RETURN
20 21	
22	Maximum Sizes for Strings
23	C type: const int (or unnamed enum)
24	Fortran type: INTEGER
25	MPI_MAX_DATAREP_STRING
26	MPI_MAX_ERROR_STRING
27	MPI_MAX_INFO_KEY
28	MPI_MAX_INFO_VAL MPI_MAX_LIBRARY_VERSION_STRING
29	MPI_MAX_DBJECT_NAME
30	MPI_MAX_OBJECT_NAME MPI_MAX_PORT_NAME
31	MPI_MAX_PROCESSOR_NAME
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

Named Predefined Datatypes	C types
C type: MPI_Datatype	
Fortran type: INTEGER	
or TYPE(MPI_Datatype)	
MPI_CHAR	char
	(treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG_INT	signed long long
MPI_LONG_LONG (as a synonym)	signed long long
MPI_SIGNED_CHAR	signed char
	(treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char
	(treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
	unsigned long long
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_AINT	MPI_Aint
MPI_COUNT	MPI_Count
MPI_COUNT MPI_OFFSET	MPI_Count MPI_Offset
MPI_OFFSET MPI_C_COMPLEX	
	float _Complex
	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	(any C type)
MPI_PACKED	(any C type)

Named Predefined Datatypes	Fortrar	n types
C type: MPI_Datatype		
Fortran type: INTEGER		
or TYPE(MPI_Datatype)		
MPI_INTEGER	INTEGEF	l
MPI_REAL	REAL	
MPI_DOUBLE_PRECISION	DOUBLE	PRECISION
MPI_COMPLEX	COMPLEX	
MPI_LOGICAL	LOGICAI	
MPI_CHARACTER	CHARACT	'ER(1)
MPI_AINT	INTEGEF	KIND=MPI_ADDRESS_KIND
_ MPI_COUNT		(KIND=MPI_COUNT_KIND)
MPI_OFFSET		(KIND=MPI_OFFSET_KIND)
MPI_BYTE		rtran type)
MPI_PACKED		tran type)
MITT_FACTED	(any 10	tran type)
Named Predefined Datatype	$\mathbf{s}^1 \mid \mathbf{C} +$	+ types
C type: MPI_Datatype		
Fortran type: INTEGER		
or TYPE(MPI_Datatype)		
MPI_CXX_BOOL	boo	
MPI_CXX_FLOAT_COMPLEX		::complex <float></float>
MPI_CXX_DOUBLE_COMPLEX		::complex <double></double>
MPI_CXX_LONG_DOUBLE_COMPL		::complex <double></double>
$^{-1}$ If an accompanying C++ comp		
MPI datatypes in this table are		6,
MFT datatypes in this table are	e not dem	lea.
Optional datatypes (H	Fortran)	Fortran types
C type: MPI_Datatype		
Fortran type: INTEGER		
• <b>-</b>		
OF ITPE(MP1 Datatvde)		
or TYPE(MPI_Datatype) MPI DOUBLE COMPLEX		DOUBLE COMPLEX
MPI_DOUBLE_COMPLEX		DOUBLE COMPLEX INTEGER*1
MPI_DOUBLE_COMPLEX MPI_INTEGER1		INTEGER*1
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2		INTEGER*1 INTEGER*2
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4		INTEGER*1 INTEGER*2 INTEGER*4
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL8 MPI_REAL16		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL8 MPI_REAL16 MPI_COMPLEX4		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4 MPI_COMPLEX8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4 COMPLEX*8
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4 MPI_COMPLEX8 MPI_COMPLEX16		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4 COMPLEX*8 COMPLEX*16
MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4 MPI_COMPLEX8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4 COMPLEX*8

Datatypes for reduction functions (C)	1
C type: MPI_Datatype	2
Fortran type: INTEGER or TYPE(MPI_Datatype)	3
MPI_FLOAT_INT	4
MPI_DOUBLE_INT	5
MPI_LONG_INT	6
MPI_2INT	7
MPI_SHORT_INT	8
MPI_LONG_DOUBLE_INT	9
Datatypes for reduction functions (Fortran)	10 11
C type: MPI_Datatype	12
Fortran type: INTEGER or TYPE(MPI_Datatype)	13
MPI_2REAL	14
MPI_2DOUBLE_PRECISION	15
MPI_2INTEGER	16
	17
Reserved communicators	18
C type: MPI_Comm	19
Fortran type: INTEGER or TYPE(MPI_Comm)	20
MPI_COMM_WORLD	21
MPI_COMM_SELF	22
	23
Communicator split type constants	24
C type: const int (or unnamed enum)	25
Fortran type: INTEGER	26
MPI_COMM_TYPE_SHARED	27
Results of communicator and group comparisons	28 29
C type: const int (or unnamed enum)	30
Fortran type: INTEGER	31
MPI_IDENT	32
MPI_CONGRUENT	33
MPI_SIMILAR	34
MPI_UNEQUAL	35
	36
Environmental inquiry info key	37
C type: MPI_Info	38
Fortran type: INTEGER or TYPE(MPI_Info)	39
MPI_INFO_ENV	40
	41
Environmental inquiry keys	42
C type: const int (or unnamed enum)	43
Fortran type: INTEGER	44
MPI_TAG_UB	45
MPI_IO	46
MPI_HOST	47
MPI_WTIME_IS_GLOBAL	48

Collective Operations
C type: MPI_Op
Fortran type: INTEGER or TYPE(MPI_Op)
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_PROD MPI_MAXLOC
MPI_MIALOC
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR
MPI_REPLACE
MPI_NO_OP
Null Handles
C/Fortran name
C type / Fortran type
MPI_GROUP_NULL
MPI_Group / INTEGER or TYPE(MPI_Group)
MPI_COMM_NULL
MPI_Comm / INTEGER or TYPE(MPI_Comm)
MPI_DATATYPE_NULL
$\texttt{MPI_Datatype} \ / \ \texttt{INTEGER} \ \text{or} \ \texttt{TYPE}(\texttt{MPI_Datatype})$
MPI_REQUEST_NULL
MPI_Request / INTEGER or TYPE(MPI_Request)
MPI_OP_NULL
MPI_Op / INTEGER or TYPE(MPI_Op)
MPI_ERRHANDLER_NULL
MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)
MPI_FILE_NULL
MPI_File / INTEGER or TYPE(MPI_File)
MPI_INFO_NULL
MPI_Info / INTEGER or TYPE(MPI_Info)
MPI_WIN_NULL
MPI_Win / INTEGER or TYPE(MPI_Win)
MPI_WIN / INTEGER OF TIPE(MPI_WIN)
<pre>MPI_Message / INTEGER or TYPE(MPI_Message)</pre>
Empty group
C type: MPI_Group
·
Fortran type: INTEGER or TYPE(MPI_Group) MPI_GROUP_EMPTY

	Topologies
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_GRAPH
	MPI_CART
	MPI_DIST_GRAPH
	Predefined functions
C/Fortran name	
C type	
/ Fortran type with	<pre>mpi module / Fortran type with mpi_f08 module</pre>
API_COMM_NULL_	COPY_FN
MPI_Comm_copy_att	tr_function
/ COMM_COPY_ATTR_	, ,
MPI_COMM_DUP_F	
MPI_Comm_copy_att	
/ COMM_COPY_ATTR_	
API_COMM_NULL_I	
MPI_Comm_delete_a	_
/ COMM_DELETE_ATT //PI_WIN_NULL_CO	· · · · · · · · · · · · · · · · · · ·
MPI_Win_Copy_attr	
/ WIN_COPY_ATTR_F	
MPI_WIN_DUP_FN	onorion / incollond (in i_win_copy_abor_incolon) )
MPI_Win_copy_att	c function
/ WIN_COPY_ATTR_F	
/ PI_WIN_NULL_DE	
MPI_Win_delete_at	tr_function
/ WIN_DELETE_ATTR	_FUNCTION / PROCEDURE(MPI_Win_delete_attr_function) 1)
MPI_TYPE_NULL_C	OPY_FN
MPI_Type_copy_att	tr_function
/ TYPE_COPY_ATTR_	, , , , , , , , , , , , , , , , , , , ,
MPI_TYPE_DUP_FN	
MPI_Type_copy_att	
/ TYPE_COPY_ATTR_	
MPI_TYPE_NULL_D	
MPI_Type_delete_a	
/ TYPE_DELETE_ATT MPI_CONVERSION_	
MPI_CONVERSION_ MPI_Datarep_conve	
-	
/ ΠΔΤΔΒΕΡ (ΟΝΙΓΕΡΟ	
/ DATAREP_CONVERS	nplementors (on page $270$ ) and advice to users (on page $270$ )
See the advice to in	nplementors (on page 270) and advice to users (on page 270) Fortran functions MPI_COMM_NULL_COPY_FN, in

1	Deprecated predefined functions
2	C/Fortran name
3	C type / Fortran type with mpi module
4	MPI_NULL_COPY_FN
5	MPI_Copy_function / COPY_FUNCTION
6	MPI_DUP_FN
7	MPI_Copy_function / COPY_FUNCTION
8	MPI_NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUNCTION
10	
11	Predefined Attribute Keys
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_APPNUM
15	MPI_LASTUSEDCODE
16	MPI_UNIVERSE_SIZE
17	MPI_WIN_BASE
18	MPI_WIN_DISP_UNIT
19	MPI_WIN_SIZE
20	MPI_WIN_CREATE_FLAVOR
21	MPI_WIN_MODEL
22	
23	MPI Window Create Flavors
24	C type: const int (or unnamed enum)
25	Fortran type: INTEGER
26	MPI_WIN_FLAVOR_CREATE
27	MPI_WIN_FLAVOR_ALLOCATE
28	MPI_WIN_FLAVOR_DYNAMIC
29	MPI_WIN_FLAVOR_SHARED
30	
31	MPI Window Models
32	C type: const int (or unnamed enum)
33	Fortran type: INTEGER
34	MPI_WIN_SEPARATE
35	MPI_WIN_UNIFIED
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	

Mode Constants
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_MODE_APPEND
MPI_MODE_CREATE
MPI_MODE_DELETE_ON_CLOSE
MPI_MODE_EXCL
MPI_MODE_NOCHECK
MPI_MODE_NOPRECEDE
MPI_MODE_NOPUT
MPI_MODE_NOSTORE
MPI_MODE_NOSUCCEED
MPI_MODE_RDONLY
MPI_MODE_RDWR
MPI_MODE_SEQUENTIAL
MPI_MODE_UNIQUE_OPEN
MPI_MODE_WRONLY
Datatype Decoding Constants
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_COMBINER_CONTIGUOUS
MPI_COMBINER_DARRAY
MPI_COMBINER_DUP
MPI_COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER
MPI_COMBINER_F90_REAL
MPI_COMBINER_HINDEXED
MPI_COMBINER_HVECTOR
MPI_COMBINER_INDEXED_BLOCK
MPI_COMBINER_HINDEXED_BLOCK MPI_COMBINER_INDEXED
MPI_COMBINER_INDEXED
MPI_COMBINER_RESIZED MPI_COMBINER_STRUCT
MPI_COMBINER_SUBARRAY
MPI_COMBINER_VECTOR
Threads Constants
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_THREAD_FUNNELED
MPI_THREAD_MULTIPLE
MPI_THREAD_SERIALIZED
MPI_THREAD_SINGLE

1	File Operation Constants, Part 1
2	C type: const MPI_Offset (or unnamed enum)
3	
4	Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)
4 5	MPI_DISPLACEMENT_CURRENT
6	File Organization Constants Dort 2
	File Operation Constants, Part 2
7	C type: const int (or unnamed enum)
8 9	Fortran type: INTEGER
9 10	
10	
12	MPI_DISTRIBUTE_DFLT_DARG
	MPI_DISTRIBUTE_NONE
13 14	MPI_ORDER_C
15	MPI_ORDER_FORTRAN
16	MPI_SEEK_CUR
17	MPI_SEEK_END
18	MPI_SEEK_SET
19	F00 Deteture Metching Constants
20	<b>F90 Datatype Matching Constants</b> C type: const int (or unnamed enum)
21	Fortran type: INTEGER
22	MPI_TYPECLASS_COMPLEX
23	MPI_TYPECLASS_COMPLEX MPI_TYPECLASS_INTEGER
24	MPI_TYPECLASS_REAL
25	
25 26	Constants Specifying Empty or Ignored Input
	Constants Specifying Empty or Ignored Input
26	C/Fortran name
26 27	C/Fortran name C type / Fortran type <sup>1</sup>
26 27 28	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL
26 27 28 29	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*)
26 27 28 29 30	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL
26 27 28 29 30 31	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*)
26 27 28 29 30 31 32	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*)
26 27 28 29 30 31 32 33	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE
26 27 28 29 30 31 32 33 34	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array
26 27 28 29 30 31 32 33 34 35	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE
26 27 28 29 30 31 32 33 34 35 36	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)
26 27 28 29 30 31 32 33 34 35 36	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*)
26 27 28 29 30 31 32 33 34 35 36 37 38	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE
26 27 28 29 30 31 32 33 34 35 36 35 36 37 38	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
26 27 28 29 30 31 32 33 34 35 36 37 38 39	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status)
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) MPI_UNWEIGHTED
26 27 28 29 30 31 32 33 34 35 36 35 36 37 38 39 40 41	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 41 42 43	C/Fortran name C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MPI_ARGV_NULL char** / array of CHARACTER*(*) MPI_ERRCODES_IGNORE int* / INTEGER array MPI_STATUSES_IGNORE MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array <sup>1</sup> Note that in Fortran these constants are not usable for initialization

C Constants Specifying Ignored Input (no Fortran)		
C type: MPI_Fint*	equivalent to Fortran	2
MPI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.h	3
MPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h	4
C type: MPI_F08_status*	equivalent to Fortran	5
MPI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08	6
MPI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08	7
~ ~ ~		8
	stants and Fortran Parameters	9
·	nacro that expands to an int value	10 11
Fortran type: INTEGER		11
MPI_SUBVERSION		12
MPI_VERSION		10
Null handles used in t	he MPI tool information interface	15
MPI_T_ENUM_NULL		16
MPI_T_enum		17
MPI_T_CVAR_HANDLE_NU	LL	18
MPI_T_cvar_handle		19
MPI_T_PVAR_HANDLE_NU	LL	20
MPI_T_pvar_handle		21
MPI_T_PVAR_SESSION_NU	ILL	22
MPI_T_pvar_session		23
		24
	e MPI tool information interface	25
C type: const int (or unn	amed enum)	26
Fortran type: INTEGER		27
MPI_T_VERBOSITY_USER_BASIC		28
MPI_T_VERBOSITY_USER		29 30
MPI_T_VERBOSITY_USER		30
MPI_T_VERBOSITY_TUNE		32
MPI_T_VERBOSITY_TUNE		33
MPI_T_VERBOSITY_TUNE	—	34
MPI_T_VERBOSITY_MPID	—	35
MPI_T_VERBOSITY_MPID	_	36
MPI_T_VERBOSITY_MPID	EV_ALL	37
		38
		39
		40
		41
		42
		43
		44
		45
		46
		47
		48

578	ANNEX A. LANGUAGE BINDINGS
	Constants to identify associations of variables
	in the MPI tool information interface
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_T_BIND_NO_OBJECT
	MPI_T_BIND_MPI_COMM
	MPI_T_BIND_MPI_DATATYPE
	MPI_T_BIND_MPI_ERRHANDLER
	MPI_T_BIND_MPI_FILE
	MPI_T_BIND_MPI_GROUP
	MPI_T_BIND_MPI_OP
	MPI_T_BIND_MPI_REQUEST
	MPI_T_BIND_MPI_WIN
	MPI_T_BIND_MPI_MESSAGE
	MPI_T_BIND_MPI_INFO
	Constants describing the scope of a control variable
	in the MPI tool information interface
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_T_SCOPE_CONSTANT
	MPI_T_SCOPE_READONLY
	MPI_T_SCOPE_LOCAL
	MPI_T_SCOPE_GROUP
	MPI_T_SCOPE_GROUP_EQ
	MPI_T_SCOPE_ALL
	MPI_T_SCOPE_ALL_EQ
	Additional constants used
	by the MPI tool information interface
	C type: MPI_T_pvar_handle
	MPI_T_PVAR_ALL_HANDLES
	Performance variables classes used by the
	MPI tool information interface
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_T_PVAR_CLASS_STATE
	MPI_T_PVAR_CLASS_LEVEL
	MPI_T_PVAR_CLASS_SIZE
	MPI_T_PVAR_CLASS_PERCENTAGE
	MPI_T_PVAR_CLASS_HIGHWATERMARK
	MPI_T_PVAR_CLASS_LOWWATERMARK
	MPI_T_PVAR_CLASS_COUNTER
	MPI_T_PVAR_CLASS_AGGREGATE
	MPI_T_PVAR_CLASS_TIMER

A.1.2 Types	1
The following are defined C type definitions, included in the file mpi.h.	2 3
/* C opaque types */	4
MPI_Aint	5
MPI_Count	6
MPI_Fint	7
MPI_Offset	8
 MPI_Status	9
 MPI_F08_status	10
	11
/* C handles to assorted structures */	12
MPI_Comm	13
MPI_Datatype	14
MPI_Errhandler	15
MPI_File	16
MPI_Group	17
MPI_Info	18
MPI_Message	19
MPI_Op	20
MPI_Request	21
MPI_Win	22
	23
/* Types for the MPI_T interface */	24
MPI_T_enum	25
MPI_T_cvar_handle	26
MPI_T_pvar_handle	27
MPI_T_pvar_session	28
	29
	30
The following are defined Fortran type definitions, included in the mpi_f08 and mpi	31
modules.	32
! Fortran opaque types in the mpi_f08 and mpi modules	33
TYPE(MPI_Status)	34
	35 36
! Fortran handles in the mpi_f08 and mpi modules	30
TYPE(MPI_Comm)	38
TYPE(MPI_Datatype)	39
TYPE(MPI_Errhandler)	40
TYPE(MPI_File)	41
TYPE(MPI_Group)	42
TYPE(MPI_Info)	43
TYPE(MPI_Op)	44
TYPE(MPI_Request)	45
TYPE(MPI_Win)	46
	47

1 A.1.3 Prototype Definitions  $\mathbf{2}$ C Bindings 3 4 The following are defined C typedefs for user-defined functions, also included in the file  $\mathbf{5}$ mpi.h. 6  $\overline{7}$ /\* prototypes for user-defined functions \*/ typedef void MPI\_User\_function(void \*invec, void \*inoutvec, int \*len, 8 9 MPI\_Datatype \*datatype); 10  $^{11}$ typedef int MPI\_Comm\_copy\_attr\_function(MPI\_Comm oldcomm, int comm\_keyval, void \*extra\_state, void \*attribute\_val\_in, 12void \*attribute\_val\_out, int \*flag); 13 14typedef int MPI\_Comm\_delete\_attr\_function(MPI\_Comm comm, 15int comm\_keyval, void \*attribute\_val, void \*extra\_state); 1617typedef int MPI\_Win\_copy\_attr\_function(MPI\_Win oldwin, int win\_keyval, 18 void \*extra\_state, void \*attribute\_val\_in, 19void \*attribute\_val\_out, int \*flag); typedef int MPI\_Win\_delete\_attr\_function(MPI\_Win win, int win\_keyval, 2021void \*attribute\_val, void \*extra\_state); 2223typedef int MPI\_Type\_copy\_attr\_function(MPI\_Datatype oldtype,  $^{24}$ int type\_keyval, void \*extra\_state, 25void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag); 26typedef int MPI\_Type\_delete\_attr\_function(MPI\_Datatype datatype, int type\_keyval, void \*attribute\_val, void \*extra\_state); 2728typedef void MPI\_Comm\_errhandler\_function(MPI\_Comm \*, int \*, ...); 29typedef void MPI\_Win\_errhandler\_function(MPI\_Win \*, int \*, ...); 30  $^{31}$ typedef void MPI\_File\_errhandler\_function(MPI\_File \*, int \*, ...); 3233 typedef int MPI\_Grequest\_query\_function(void \*extra\_state, 34 MPI\_Status \*status); typedef int MPI\_Grequest\_free\_function(void \*extra\_state); 35typedef int MPI\_Grequest\_cancel\_function(void \*extra\_state, int complete); 36 37 typedef int MPI\_Datarep\_extent\_function(MPI\_Datatype datatype, 3839MPI\_Aint \*file\_extent, void \*extra\_state); typedef int MPI\_Datarep\_conversion\_function(void \*userbuf, 4041 MPI\_Datatype datatype, int count, void \*filebuf, 42MPI\_Offset position, void \*extra\_state); 43 44Fortran 2008 Bindings with the mpi\_f08 Module 45The callback prototypes when using the Fortran mpi\_f08 module are shown below: 46The user-function argument to MPI\_Op\_create should be declared according to: 47ABSTRACT INTERFACE 48

```
1
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)
                                                                                   2
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    3
      TYPE(C_PTR), VALUE :: invec, inoutvec
      INTEGER :: len
                                                                                   4
      TYPE(MPI_Datatype) :: datatype
                                                                                   5
                                                                                   6
    The copy and delete function arguments to MPI_Comm_create_keyval should be de-
                                                                                    7
clared according to:
                                                                                    8
ABSTRACT INTERFACE
                                                                                   9
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
                                                                                   10
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                   11
      TYPE(MPI_Comm) :: oldcomm
                                                                                   12
      INTEGER :: comm_keyval, ierror
                                                                                   13
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   14
      attribute_val_out
                                                                                   15
      LOGICAL :: flag
                                                                                   16
                                                                                   17
ABSTRACT INTERFACE
                                                                                   18
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
                                                                                   19
  attribute_val, extra_state, ierror) BIND(C)
      TYPE(MPI_Comm) :: comm
                                                                                   20
                                                                                   21
      INTEGER :: comm_keyval, ierror
                                                                                   22
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                   23
   The copy and delete function arguments to MPI_Win_create_keyval should be declared
                                                                                   24
according to:
                                                                                   25
ABSTRACT INTERFACE
                                                                                   26
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
                                                                                   27
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                   28
      TYPE(MPI_Win) :: oldwin
                                                                                   29
      INTEGER :: win_keyval, ierror
                                                                                   30
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   31
      attribute_val_out
                                                                                   32
      LOGICAL :: flag
                                                                                   33
                                                                                   34
ABSTRACT INTERFACE
                                                                                   35
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
                                                                                   36
  extra_state, ierror) BIND(C)
                                                                                   37
      TYPE(MPI_Win) :: win
                                                                                   38
      INTEGER :: win_keyval, ierror
                                                                                   39
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                   40
   The copy and delete function arguments to MPI_Type_create_keyval should be declared
                                                                                   41
according to:
                                                                                   42
ABSTRACT INTERFACE
                                                                                   43
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                   44
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                   45
      TYPE(MPI_Datatype) :: oldtype
                                                                                   46
      INTEGER :: type_keyval, ierror
                                                                                   47
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   48
```

1attribute\_val\_out  $\mathbf{2}$ LOGICAL :: flag 3 ABSTRACT INTERFACE 4 SUBROUTINE MPI\_Type\_delete\_attr\_function(datatype, type\_keyval, 5attribute\_val, extra\_state, ierror) BIND(C) 6 TYPE(MPI\_Datatype) :: datatype 7 INTEGER :: type\_keyval, ierror 8 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: attribute\_val, extra\_state 9 10 The handler-function argument to MPI\_Comm\_create\_errhandler should be declared 11 like this: 12ABSTRACT INTERFACE 13SUBROUTINE MPI\_Comm\_errhandler\_function(comm, error\_code) BIND(C) 14TYPE(MPI\_Comm) :: comm 15INTEGER :: error\_code 16 The handler-function argument to MPI\_Win\_create\_errhandler should be declared like 17this: 18 ABSTRACT INTERFACE 19 SUBROUTINE MPI\_Win\_errhandler\_function(win, error\_code) BIND(C) 20TYPE(MPI\_Win) :: win 21INTEGER :: error\_code 22 23The handler-function argument to MPI\_File\_create\_errhandler should be declared like 24this: 25ABSTRACT INTERFACE 26SUBROUTINE MPI\_File\_errhandler\_function(file, error\_code) BIND(C) 27TYPE(MPI\_File) :: file 28INTEGER :: error\_code 29 The query, free, and cancel function arguments to MPI\_Grequest\_start should be de-30 clared according to: 31 ABSTRACT INTERFACE 32 SUBROUTINE MPI\_Grequest\_query\_function(extra\_state, status, ierror) 33 BIND(C) 34 TYPE(MPI\_Status) :: status 35 INTEGER :: ierror 36 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 37 38ABSTRACT INTERFACE 39 SUBROUTINE MPI\_Grequest\_free\_function(extra\_state, ierror) BIND(C) 40INTEGER :: ierror 41 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 42ABSTRACT INTERFACE 43 SUBROUTINE MPI\_Grequest\_cancel\_function(extra\_state, complete, ierror) 44 BIND(C) 45 INTEGER :: ierror 46INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 47 LOGICAL :: complete 48

```
1
    The extent and conversion function arguments to MPI_Register_datarep should be de-
                                                                                      2
clared according to:
                                                                                      3
ABSTRACT INTERFACE
                                                                                      4
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
  ierror) BIND(C)
                                                                                      5
                                                                                      6
      TYPE(MPI_Datatype) :: datatype
                                                                                      7
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
                                                                                      8
      INTEGER :: ierror
                                                                                      9
ABSTRACT INTERFACE
                                                                                     10
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
                                                                                     11
  filebuf, position, extra_state, ierror) BIND(C)
                                                                                     12
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                     13
      TYPE(C_PTR), VALUE :: userbuf, filebuf
                                                                                     14
      TYPE(MPI_Datatype) :: datatype
                                                                                     15
      INTEGER :: count, ierror
                                                                                     16
      INTEGER(KIND=MPI_OFFSET_KIND) :: position
                                                                                     17
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                     18
                                                                                     19
                                                                                     <sup>20</sup> ticketWG.
Fortran Bindings with the deprecated mpif.h or the mpi Module
                                                                                     21
With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
                                                                                     22
subroutines should be declared.
                                                                                     23
    The user-function argument to MPI_OP_CREATE should be declared like this:
                                                                                     24
                                                                                     25
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
                                                                                     26
   <type> INVEC(LEN), INOUTVEC(LEN)
                                                                                     27
   INTEGER LEN, DATATYPE
                                                                                     28
                                                                                     29
   The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
                                                                                     30
declared like these:
                                                                                     31
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                                                                                     32
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     33
   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
                                                                                     34
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     35
              ATTRIBUTE_VAL_OUT
                                                                                     36
   LOGICAL FLAG
                                                                                     37
                                                                                     38
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                                                                                     39
             EXTRA_STATE, IERROR)
                                                                                     40
   INTEGER COMM, COMM_KEYVAL, IERROR
                                                                                     41
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     42
                                                                                     43
    The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be
                                                                                     44
declared like these:
                                                                                     45
                                                                                     46
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                                                                                     47
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     48
```

1INTEGER OLDWIN, WIN\_KEYVAL, IERROR  $\mathbf{2}$ INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 3 ATTRIBUTE\_VAL\_OUT 4 LOGICAL FLAG 56 SUBROUTINE WIN\_DELETE\_ATTR\_FUNCTION(WIN, WIN\_KEYVAL, ATTRIBUTE\_VAL, 7EXTRA\_STATE, IERROR) 8 INTEGER WIN, WIN\_KEYVAL, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL, EXTRA\_STATE 9 10 The copy and delete function arguments to MPI\_TYPE\_CREATE\_KEYVAL should be 11 declared like these: 1213 SUBROUTINE TYPE\_COPY\_ATTR\_FUNCTION(OLDTYPE, TYPE\_KEYVAL, EXTRA\_STATE, 14ATTRIBUTE\_VAL\_IN, ATTRIBUTE\_VAL\_OUT, FLAG, IERROR) 15INTEGER OLDTYPE, TYPE\_KEYVAL, IERROR 16INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE, 17ATTRIBUTE\_VAL\_IN, ATTRIBUTE\_VAL\_OUT 18 LOGICAL FLAG 19 20SUBROUTINE TYPE\_DELETE\_ATTR\_FUNCTION(DATATYPE, TYPE\_KEYVAL, ATTRIBUTE\_VAL, 21EXTRA\_STATE, IERROR) 22 INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 23INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL, EXTRA\_STATE 2425The handler-function argument to MPI\_COMM\_CREATE\_ERRHANDLER should be de-26clared like this: 2728SUBROUTINE COMM\_ERRHANDLER\_FUNCTION(COMM, ERROR\_CODE) 29 INTEGER COMM, ERROR\_CODE 30  $^{31}$ The handler-function argument to MPI\_WIN\_CREATE\_ERRHANDLER should be de-32 clared like this: 33 34SUBROUTINE WIN\_ERRHANDLER\_FUNCTION(WIN, ERROR\_CODE) 35 INTEGER WIN, ERROR\_CODE 36 37 The handler-function argument to MPI\_FILE\_CREATE\_ERRHANDLER should be de-38 clared like this: 39 40SUBROUTINE FILE\_ERRHANDLER\_FUNCTION(FILE, ERROR\_CODE) 41 INTEGER FILE, ERROR\_CODE 42The query, free, and cancel function arguments to MPI\_GREQUEST\_START should be 43 declared like these: 4445SUBROUTINE GREQUEST\_QUERY\_FUNCTION(EXTRA\_STATE, STATUS, IERROR) 46INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR 47INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE 48

```
2
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                     5
                                                                                     6
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
                                                                                     10
   The extent and conversion function arguments to MPI_REGISTER_DATAREP should
                                                                                     11
be declared like these:
                                                                                     12
                                                                                     13
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
                                                                                     14
    INTEGER DATATYPE, IERROR
                                                                                     15
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
                                                                                     16
                                                                                     17
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
                                                                                     18
             POSITION, EXTRA_STATE, IERROR)
                                                                                     19
    <TYPE> USERBUF(*), FILEBUF(*)
                                                                                     20
    INTEGER COUNT, DATATYPE, IERROR
                                                                                     21
    INTEGER(KIND=MPI_OFFSET_KIND) POSITION
                                                                                     22
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                     23
                                                                                     24
A.1.4 Deprecated Prototype Definitions
                                                                                     25
                                                                                     26
The following are defined C typedefs for deprecated user-defined functions, also included in
                                                                                     27
the file mpi.h.
                                                                                     28
                                                                                     29
/* prototypes for user-defined functions */
                                                                                     30
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                                                                                     31
               void *extra_state, void *attribute_val_in,
                                                                                     32
               void *attribute_val_out, int *flag);
                                                                                     33
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
                                                                                     34
               void *attribute_val, void *extra_state);
                                                                                     35
                                                                                     36
    The following are deprecated Fortran user-defined callback subroutine prototypes. The
                                                                                     37
deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-
                                                                                     38
clared like these:
                                                                                     39
                                                                                     40
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
                                                                                     41
                ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
                                                                                     42
   INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     43
         ATTRIBUTE_VAL_OUT, IERR
                                                                                     44
   LOGICAL FLAG
                                                                                     45
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
                                                                                     46
                                                                                     47
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
                                                                                     48
```

# <sup>1</sup> A.1.5 Info Keys

- $\frac{2}{3}$  The following info keys are reserved. They are strings.
- 4 access\_style
- 5 appnum
- 6 arch
- 7 cb\_block\_size
- 8 cb\_buffer\_size
- 9 cb\_nodes
- 10 chunked\_item
- 11 chunked\_size
- 12 chunked
- 13 collective\_buffering
- 14 file\_perm
- 15 filename
- 16 file
- 17 host
- 18 io\_node\_list
- 19 ip\_address
- 20 ip\_port
- <sup>21</sup> nb\_proc
- 22 no\_locks
- 23 num\_io\_nodes
- 24 path
- 25 soft
- 26 striping\_factor
- 27 striping\_unit
- 28 wdir
- 29
- 30

```
<sup>31</sup> A.1.6 Info Values
```

 $^{32}_{33}$  The following info values are reserved. They are strings.

<sub>34</sub> false

- 35 random
- <sub>36</sub> read\_mostly
- 37 read\_once
- 38 reverse\_sequential
- 39 sequential
- 40 true
- 41 write\_mostly
- 42 write\_once
- 43
- 44
- 45
- 46
- 47
- 48

A.2 C Bindings	1
A.2.1 Point-to-Point Communication C Bindings	2 3
<pre>int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	4 5 6
<pre>int MPI_Bsend_init(const void* buf, int count, MPI_Datatype datatype,</pre>	7 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 15
<pre>int MPI_Ibsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	16 17 18
int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status)	19 20 21
int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)	21 22 23
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)	24 25 26
<pre>int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,</pre>	20 27 28
<pre>int MPI_Irsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	29 30 31
<pre>int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	32 33
<pre>int MPI_Issend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	34 35 36
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)	37 38
int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)	39 40 41
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	42
<pre>int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,</pre>	43 44 45
<pre>int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,</pre>	46 47 48

1 int MPI\_Request\_free(MPI\_Request \*request)  $\mathbf{2}$ int MPI\_Request\_get\_status(MPI\_Request request, int \*flag, 3 MPI\_Status \*status) 4  $\mathbf{5}$ int MPI\_Rsend(const void\* buf, int count, MPI\_Datatype datatype, int dest, 6 int tag, MPI\_Comm comm) 7int MPI\_Rsend\_init(const void\* buf, int count, MPI\_Datatype datatype, 8 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 9 10int MPI\_Send(const void\* buf, int count, MPI\_Datatype datatype, int dest, 11int tag, MPI\_Comm comm) 12int MPI\_Send\_init(const void\* buf, int count, MPI\_Datatype datatype, 13int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 1415int MPI\_Sendrecv(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 16int dest, int sendtag, void \*recvbuf, int recvcount, 17MPI\_Datatype recvtype, int source, int recvtag, MPI\_Comm comm, 18 MPI\_Status \*status) 19int MPI\_Sendrecv\_replace(void\* buf, int count, MPI\_Datatype datatype, 20int dest, int sendtag, int source, int recvtag, MPI\_Comm comm, 21MPI\_Status \*status) 2223int MPI\_Ssend(const void\* buf, int count, MPI\_Datatype datatype, int dest,  $^{24}$ int tag, MPI\_Comm comm) 25int MPI\_Ssend\_init(const void\* buf, int count, MPI\_Datatype datatype, 26int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 2728int MPI\_Start(MPI\_Request \*request) 29int MPI\_Startall(int count, MPI\_Request array\_of\_requests[]) 30  $^{31}$ int MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_Status \*status) 32int MPI\_Test\_cancelled(const MPI\_Status \*status, int \*flag) 33 34int MPI\_Testall(int count, MPI\_Request array\_of\_requests[], int \*flag, 35MPI\_Status array\_of\_statuses[]) 36 37int MPI\_Testany(int count, MPI\_Request array\_of\_requests[], int \*index, 38int \*flag, MPI\_Status \*status) 39int MPI\_Testsome(int incount, MPI\_Request array\_of\_requests[], 40int \*outcount, int array\_of\_indices[], 41 MPI\_Status array\_of\_statuses[]) 4243int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status) 44int MPI\_Waitall(int count, MPI\_Request array\_of\_requests[], 45MPI\_Status array\_of\_statuses[]) 4647int MPI\_Waitany(int count, MPI\_Request array\_of\_requests[], int \*index, 48

ANNEX A. LANGUAGE BINDINGS SUMMARY

	MPI_Status *status)	1
int	MPI_Waitsome(int incount, MPI_Request array_of_requests[],	2
1110	int *outcount, int array_of_indices[],	3
	MPI_Status array_of_statuses[])	4
		5 6
		7
A.2.	2 Datatypes C Bindings	8
int	<pre>MPI_Get_address(const void *location, MPI_Aint *address)</pre>	9
int	<pre>MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>	10 11 12
int	<pre>MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype, MPI_Count *count)</pre>	12 13 14
int	MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype,	15
1110	void *outbuf, int outsize, int *position, MPI_Comm comm)	16
		17
int	<pre>MPI_Pack_external(const char datarep[], const void *inbuf, int incount,</pre>	18
	MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,	19
	MPI_Aint *position)	20 21
int	<pre>MPI_Pack_external_size(const char datarep[], int incount,</pre>	21
	MPI_Datatype datatype, MPI_Aint *size)	22
int	MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,	24
1110	int *size)	25
		26
int	MPI_Type_commit(MPI_Datatype *datatype)	27
int	MPI_Type_contiguous(int count, MPI_Datatype oldtype,	28
	MPI_Datatype *newtype)	29
in+	MDI Tupo create darray(int size int rank int ndima const	30
THC	<pre>MPI_Type_create_darray(int size, int rank, int ndims, const</pre>	31
	int array_of_dargs[], const int array_of_distributes[], const int array_of_dargs[], const int array_of_psizes[], int order,	32
	MPI_Datatype oldtype, MPI_Datatype *newtype)	33
		34
int	<pre>MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],</pre>	35
	<pre>const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,</pre>	36 37
	MPI_Datatype *newtype)	38
int	MPI_Type_create_hindexed_block(int count, int blocklength, const	39
	MPI_Aint array_of_displacements[], MPI_Datatype oldtype,	40
	MPI_Datatype *newtype)	41
in+	MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,	42
THE	MPI_Datatype oldtype, MPI_Datatype *newtype)	43
	In I_Datatype oratype, In I_Datatype *newtype)	44
int	MPI_Type_create_indexed_block(int count, int blocklength, const	45
	<pre>int array_of_displacements[], MPI_Datatype oldtype,</pre>	46
	MPI_Datatype *newtype)	47
		48

1 2	int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_ extent, MPI_Datatype *newtype)	Aint
3 4 5 6	<pre>int MPI_Type_create_struct(int count, const int array_of_blocklengt</pre>	:hs[],
7 8 9	<pre>int MPI_Type_create_subarray(int ndims, const int array_of_sizes[] int array_of_subsizes[], const int array_of_starts[], order, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	
10 11	int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)	
12	int MPI_Type_free(MPI_Datatype *datatype)	
13 14 15 16 17	<pre>int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,</pre>	tegers[],
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, MPI_Aint *extent)</pre>	
23 24 25	<pre>int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb, MPI_Count *extent)</pre>	
26 27	int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true MPI_Aint *true_extent)	_1b,
28 29 30	<pre>int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *tr MPI_Count *true_extent)</pre>	rue_lb,
31 32 33	<pre>int MPI_Type_indexed(int count, const int array_of_blocklengths[],</pre>	const
34 35	int MPI_Type_size(MPI_Datatype datatype, int *size)	
36	int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)	
37 38 39	int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)	
40 41	int MPI_Unpack(const void* inbuf, int insize, int *position, void • int outcount, MPI_Datatype datatype, MPI_Comm comm)	∗outbuf,
42 43 44 45 46 47	int MPI_Unpack_external(const char datarep[], const void *inbuf, MPI_Aint insize, MPI_Aint *position, void *outbuf, int outcount, MPI_Datatype datatype)	
48		

A.2.	3 Collective Communication C Bindings	1
int	MPI_Allgather(const void* sendbuf, int sendcount,	2 3
	MPI_Datatype sendtype, void* recvbuf, int recvcount,	4
	MPI_Datatype recvtype, MPI_Comm comm)	5
int	MPI_Allgatherv(const void* sendbuf, int sendcount,	6
	<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm)</pre>	7 8
		9
int	MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	10 11
int	<pre>MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>	12 13 14
int	MPI_Alltoallv(const void* sendbuf, const int sendcounts[], const	15 16
	<pre>int sdispls[], MPI_Datatype sendtype, void* recvbuf, const</pre>	17
	<pre>int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)</pre>	18 19
		20
IIIC	<pre>MPI_Alltoallw(const void* sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[], void* recvbuf,</pre>	21
	<pre>const int recvcounts[], const int rdispls[], const</pre>	22
	<pre>MPI_Datatype recvtypes[], MPI_Comm comm)</pre>	23 24
int	MPI_Barrier(MPI_Comm comm)	25
int	MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,	26
	MPI_Comm comm)	27 28
int	MPI_Exscan(const void* sendbuf, void* recvbuf, int count,	29
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	30
int	MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	31 32
	<pre>void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MDL Gamma samm)</pre>	33
	MPI_Comm comm)	34
int	MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	35
	<pre>void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	36 37
		38
IUC	<pre>MPI_Iallgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount,</pre>	39
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	$40 \\ 41$
int	MPI_Iallgatherv(const void* sendbuf, int sendcount,	41 42
	MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],	43
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	44
	MPI_Request* request)	$45 \\ 46$
int	MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,	47
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	48

1		MPI_Request *request)
2 3 4 5	int MPI_Iall	toall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
6 7 8 9 10	int MPI_Iall	<pre>toallv(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>
11 12 13 14	int MPI_Iall	<pre>toallw(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>
15 16	int MPI_Ibar	rier(MPI_Comm comm, MPI_Request *request)
17 18	int MPI_Ibca	st(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)
19 20 21 22	int MPI_Iexs	<pre>can(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>
23 24 25	int MPI_Igat	<pre>her(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
26 27 28 29 30	int MPI_Igat	<pre>herv(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>
31 32 33	int MPI_Ired	uce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Request *request)
34 35 36 37	int MPI_Ired	uce_scatter(const void* sendbuf, void* recvbuf, const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
38 39 40	int MPI_Ired	uce_scatter_block(const void* sendbuf, void* recvbuf, int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
41 42 43 44	int MPI_Isca	n(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
45 46 47 48	int MPI_Isca	<pre>tter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>

int	<pre>MPI_Iscatterv(const void* sendbuf, const int sendcounts[], const</pre>	1 2 3 4
int	MPI_Op_commutative(MPI_Op op, int *commute)	5 6
int	MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)	7
int	MPI_Op_free(MPI_Op *op)	8 9
		9 10
THC	<pre>MPI_Reduce(const void* sendbuf, void* recvbuf, int count,</pre>	11
int	MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,	12
1110	MPI_Datatype datatype, MPI_Op op)	13 14
int	MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const	15
1110	int recvcounts[], MPI_Datatype datatype, MPI_Op op,	16
	MPI_Comm comm)	17
int	<pre>MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>	18 19
	int recvcount, MPI_Datatype datatype, MPI_Op op,	20
	MPI_Comm comm)	21
int	MPI_Scan(const void* sendbuf, void* recvbuf, int count,	22
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	23 24
int	MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	25
	<pre>void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>	26 27
·		28
int	<pre>MPI_Scatterv(const void* sendbuf, const int sendcounts[], const</pre>	29
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	30
		31 32
Δ2	4 Groups, Contexts, Communicators, and Caching C Bindings	33
		34
int	MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,	35
	<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	36
int	MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,	37 38
	<pre>void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	39
		40
int	<pre>MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void</pre>	41
		42
int	<pre>MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)</pre>	43 44
int	MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)	45
int	MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,	46
	MPI_Comm *newcomm)	47
		48

1 2 3	int	<pre>MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,</pre>
4 5	int	<pre>MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)</pre>
6	int	MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
7 8	int	MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
9	int	MPI_Comm_free(MPI_Comm *comm)
10 11	int	MPI_Comm_free_keyval(int *comm_keyval)
12 13 14	int	<pre>MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,</pre>
15	int	MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
16 17	int	<pre>MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)</pre>
18	int	MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
19 20	int	MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
21	int	MPI_Comm_rank(MPI_Comm comm, int *rank)
22 23	int	MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
24	int	MPI_Comm_remote_size(MPI_Comm comm, int *size)
25 26	int	<pre>MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)</pre>
27	int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
28 29	int	MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
30	int	MPI_Comm_size(MPI_Comm comm, int *size)
31 32	int	MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
33 34	int	<pre>MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,</pre>
35 36	int	MPI_Comm_test_inter(MPI_Comm comm, int *flag)
37	int	<pre>MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)</pre>
38 39	int	MPI_Group_difference(MPI_Group group1, MPI_Group group2,
40		MPI_Group *newgroup)
41 42 43	int	<pre>MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>
44	int	MPI_Group_free(MPI_Group *group)
45 46 47	int	<pre>MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>
48		

int	<pre>MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	1
int	<pre>MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>	3 4 5
int	<pre>MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>	6 7 8
int	MPI_Group_rank(MPI_Group group, int *rank)	8 9
int	MPI_Group_size(MPI_Group group, int *size)	10 11
int	<pre>MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[], MPI_Group group2, int ranks2[])</pre>	11 12 13
int	<pre>MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	14 15
int	<pre>MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)</pre>	16 17 18 19
int	MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)	20 21 22
int	<pre>MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	23 24 25
int	<pre>MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	26 27 28 29
int	<pre>MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void      *attribute_val, void *extra_state)</pre>	30 31
int	<pre>MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	32 33 34 35
int	MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)	36
int	MPI_Type_free_keyval(int *type_keyval)	37 38
int	<pre>MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void      *attribute_val, int *flag)</pre>	39 40
int	<pre>MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int</pre>	41 42 43
int	<pre>MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,</pre>	44 45
int	<pre>MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)</pre>	46 47 48

1 2	int	<pre>MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,</pre>
3 4 5	int	<pre>MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
6 7	int	<pre>MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void      *attribute_val, void *extra_state)</pre>
8 9 10 11	int	<pre>MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,</pre>
12	int	MPI_Win_delete_attr(MPI_Win win, int win_keyval)
13 14	int	MPI_Win_free_keyval(int *win_keyval)
15 16 17	int	<pre>MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>
18	int	MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
19	int	MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
20 21 22	int	MPI_Win_set_name(MPI_Win win, const char *win_name)
23 24	A.2.	5 Process Topologies C Bindings
25	int	<pre>MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])</pre>
26 27 28	int	<pre>MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const</pre>
29 30	int	<pre>MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>
31 32 33	int	<pre>MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[], const int periods[], int *newrank)</pre>
34	int	<pre>MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>
35 36 37	int	<pre>MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>
38	int	MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
39 40	int	MPI_Cartdim_get(MPI_Comm comm, int *ndims)
41	int	MPI_Dims_create(int nnodes, int ndims, int dims[])
42 43 44 45 46	int	<pre>MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>
47 48	int	<pre>MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree, const</pre>

	<pre>int sources[], const int sourceweights[], int outdegree, const int destinations[], const int destweights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)</pre>	1 2 3
int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	4 5 6 7
int	int *outdegree, int *weighted)	8 9 10
int	MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],	11 11 12
int	<pre>int edges[])</pre>	13 14 15
int	MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], const	16 16
int	<pre>MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	18 19 20
int	NDT Grant maintheast and (NDT Grant and intersuch intersuch intersuch)	20 21
int	MP1_Graphdims_get(MP1_Comm comm, int *nnodes, int *nedges)	22 23
int	MPI_Datatype sendtype, void* recvbuf, int recvcount,	24 25 26
int	<pre>MPI_Ineighbor_allgatherv(const void* sendbuf, int sendcount,</pre>	27 28 29 30 31
int	<pre>MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,</pre>	32 33 34
int	<pre>MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>	35 36 37 38 39
int	<pre>MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>	40 41 42 43 44
int	<pre>MPI_Neighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm_comm)</pre>	45 46 47 48

<pre>int MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount,</pre>
<pre>int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype             sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,             MPI_Comm comm)</pre>
<pre>int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>
<pre>int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>
<pre>int MPI_Topo_test(MPI_Comm comm, int *status)</pre>
A.2.6 MPI Environmental Management C Bindings
double MPI_Wtick(void)
double MPI_Wtime(void)
int MPI_Abort(MPI_Comm comm, int errorcode)
<pre>int MPI_Add_error_class(int *errorclass)</pre>
int MPI_Add_error_code(int errorclass, int *errorcode)
<pre>int MPI_Add_error_string(int errorcode, const char *string)</pre>
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)</pre>
<pre>int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function     *comm_errhandler_fn, MPI_Errhandler *errhandler)</pre>
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>
<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
<pre>int MPI_File_create_errhandler(MPI_File_errhandler_function</pre>
<pre>*file_errhandler_fn, MPI_Errhandler *errhandler)</pre>
<pre>int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)</pre>

<pre>int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)</pre>	1
<pre>int MPI_Finalize(void)</pre>	2 3
int MPI_Finalized(int *flag)	4
-	5
<pre>int MPI_Free_mem(void *base)</pre>	6
<pre>int MPI_Get_library_version(char *version, int *resultlen)</pre>	7 8
<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>	9
<pre>int MPI_Get_version(int *version, int *subversion)</pre>	10
<pre>int MPI_Init(int *argc, char ***argv)</pre>	11 12
int MPI_Initialized(int *flag)	13
<pre>int MPI_Win_call_errhandler(MPI_Win win, int errorcode)</pre>	14
	15 16
<pre>int MPI_Win_create_errhandler(MPI_Win_errhandler_function</pre>	17
	18
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)	19
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)	20 21
	22
A.2.7 The Info Object C Bindings	23
<pre>int MPI_Info_create(MPI_Info *info)</pre>	24 25
int MPI_Info_delete(MPI_Info info, const char *key)	26
<pre>int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)</pre>	27
-	28 29
<pre>int MPI_Info_free(MPI_Info *info)</pre>	30
<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>	31
<pre>int *flag)</pre>	32
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)	33 34
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)	35
int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,	36
int *flag)	37
int MPI_Info_set(MPI_Info info, const char *key, const char *value)	38 39
	40
A.2.8 Process Creation and Management C Bindings	41
<pre>int MPI_Close_port(const char *port_name)</pre>	42 43
	44
<pre>int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,</pre>	45
	46 47
<pre>int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,</pre>	47

1	int MPI_Comm_di	<pre>sconnect(MPI_Comm *comm)</pre>
2 3	int MPI_Comm_ge	t_parent(MPI_Comm *parent)
4	int MPI_Comm_jo	in(int fd, MPI_Comm *intercomm)
5 6 7 8	MF	<pre>awn(const char *command, char *argv[], int maxprocs, PI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, nt array_of_errcodes[])</pre>
9 10 11 12	ch MF	<pre>pawn_multiple(int count, char *array_of_commands[], har **array_of_argv[], const int array_of_maxprocs[], const PI_Info array_of_info[], int root, MPI_Comm comm, PI_Comm *intercomm, int array_of_errcodes[])</pre>
13 14 15	-	name(const char *service_name, MPI_Info info, nar *port_name)
16 17	int MPI_Open_po	rt(MPI_Info info, char *port_name)
18 19		_name(const char *service_name, MPI_Info info, const nar *port_name)
20 21 22	-	sh_name(const char *service_name, MPI_Info info, const nar *port_name)
23 24	A 2 0 One Sider	Communications C Bindings
	A.2.9 Olle-Sided	Communications C Dindings
25 26 27 28 29	int MPI_Accumul MF MF	ate(const void *origin_addr, int origin_count, PI_Datatype origin_datatype, int target_rank, PI_Aint target_disp, int target_count, PI_Datatype target_datatype, MPI_Op op, MPI_Win win)
26 27 28	int MPI_Accumul MF MF MF int MPI_Compare vc	ate(const void *origin_addr, int origin_count, PI_Datatype origin_datatype, int target_rank, PI_Aint target_disp, int target_count,
26 27 28 29 30 31	int MPI_Accumul MF MF int MPI_Compare vc MF int MPI_Fetch_a MF	<pre>ate(const void *origin_addr, int origin_count, PI_Datatype origin_datatype, int target_rank, PI_Aint target_disp, int target_count, PI_Datatype target_datatype, MPI_Op op, MPI_Win win) a_and_swap(const void *origin_addr, const void *compare_addr, oid *result_addr, MPI_Datatype datatype, int target_rank,</pre>
26 27 28 29 30 31 32 33 34 35	int MPI_Accumul MF MF int MPI_Compare vc MF int MPI_Fetch_a MF int MPI_Get(voi MF MF	<pre>ate(const void *origin_addr, int origin_count, PI_Datatype origin_datatype, int target_rank, PI_Aint target_disp, int target_count, PI_Datatype target_datatype, MPI_Op op, MPI_Win win) e_and_swap(const void *origin_addr, const void *compare_addr, oid *result_addr, MPI_Datatype datatype, int target_rank, PI_Aint target_disp, MPI_Win win) nd_op(const void *origin_addr, void *result_addr, PI_Datatype datatype, int target_rank, MPI_Aint target_disp,</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39	<pre>int MPI_Accumul MF MF MF int MPI_Compare vc MF int MPI_Fetch_a MF int MPI_Get(voi MF MF int MPI_Get_acc MF int MPI_Get_acc MF ir ir</pre>	<pre>ate(const void *origin_addr, int origin_count, PI_Datatype origin_datatype, int target_rank, PI_Aint target_disp, int target_count, PI_Datatype target_datatype, MPI_Op op, MPI_Win win) a_and_swap(const void *origin_addr, const void *compare_addr, oid *result_addr, MPI_Datatype datatype, int target_rank, PI_Aint target_disp, MPI_Win win) and_op(const void *origin_addr, void *result_addr, PI_Datatype datatype, int target_rank, MPI_Aint target_disp, PI_Op op, MPI_Win win) d *origin_addr, int origin_count, PI_Datatype origin_datatype, int target_rank, PI_Aint target_disp, int target_rank, PI_Datatype origin_datatype, int target_rank,</pre>

	origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)	1 2
int	MPI_Raccumulate(const void *origin_addr, int origin_count,	3
1110	MPI_Datatype origin_datatype, int target_rank,	4
	MPI_Aint target_disp, int target_count,	5
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,	6
	MPI_Request *request)	7
		8
int	MPI_Rget(void *origin_addr, int origin_count,	9 10
	MPI_Datatype origin_datatype, int target_rank,	10
	MPI_Aint target_disp, int target_count,	12
	MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)	13
	Mri_nequest *iequest)	14
int	<pre>MPI_Rget_accumulate(const void *origin_addr, int origin_count,</pre>	15
	<pre>MPI_Datatype origin_datatype, void *result_addr,</pre>	16
	<pre>int result_count, MPI_Datatype result_datatype,</pre>	17
	<pre>int target_rank, MPI_Aint target_disp, int target_count,</pre>	18
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,	19
	MPI_Request *request)	20
int	<pre>MPI_Rput(const void *origin_addr, int origin_count,</pre>	21
	MPI_Datatype origin_datatype, int target_rank,	22
	MPI_Aint target_disp, int target_count,	23
	MPI_Datatype target_datatype, MPI_Win win,	24
	MPI_Request *request)	25 26
int	MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,	20
	MPI_Comm comm, void *baseptr, MPI_Win *win)	28
int	MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,	29
	MPI_Comm comm, void *baseptr, MPI_Win *win)	30
		31
int	<pre>MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)</pre>	32
int	MPI_Win_complete(MPI_Win win)	33 34
int	MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,	35
	MPI_Comm comm, MPI_Win *win)	36
int	MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)	37 38
int	MPI_Win_detach(MPI_Win win, const void *base)	39
int	MPI_Win_fence(int assert, MPI_Win win)	40
int	MPI_Win_flush(int rank, MPI_Win win)	41 42
		43
TUL	MPI_Win_flush_all(MPI_Win win)	44
int	MPI_Win_flush_local(int rank, MPI_Win win)	45 46
int	MPI_Win_flush_local_all(MPI_Win win)	40 47
		48

```
1
     int MPI_Win_free(MPI_Win *win)
\mathbf{2}
     int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
3
4
     int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
5
     int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
6
7
     int MPI_Win_lock_all(int assert, MPI_Win win)
8
     int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
9
10
     int MPI_Win_set_info(MPI_Win win, MPI_Info info)
11
     int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
12
                   int *disp_unit, void *baseptr)
13
14
     int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
15
     int MPI_Win_sync(MPI_Win win)
16
17
     int MPI_Win_test(MPI_Win win, int *flag)
18
     int MPI_Win_unlock(int rank, MPI_Win win)
19
20
     int MPI_Win_unlock_all(MPI_Win win)
21
     int MPI_Win_wait(MPI_Win win)
22
23
^{24}
     A.2.10 External Interfaces C Bindings
25
26
     int MPI_Grequest_complete(MPI_Request request)
27
     int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
28
                   MPI_Grequest_free_function *free_fn,
29
                   MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
30
                   MPI_Request *request)
^{31}
32
     int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
33
     int MPI_Is_thread_main(int *flag)
34
35
     int MPI_Query_thread(int *provided)
36
     int MPI_Status_set_cancelled(MPI_Status *status, int flag)
37
38
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
39
                   int count)
40
     int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
41
                   MPI_Count count)
42
43
44
     A.2.11 I/O C Bindings
45
46
     int MPI_File_close(MPI_File *fh)
47
     int MPI_File_delete(const char *filename, MPI_Info info)
48
```

<pre>int MPI_File_get_amode(MPI_File fh, int *amode)</pre>	1
<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>	2 3
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,	4
MPI_Offset *disp)	5
int MPI_File_get_group(MPI_File fh, MPI_Group *group)	6 7
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)	8
ů – Elektrik Alektrik – Elektrik	9
<pre>int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)</pre>	10 11
<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>	12
<pre>int MPI_File_get_size(MPI_File fh, MPI_Offset *size)</pre>	13
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,	14
MPI_Aint *extent)	15 16
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,	17
MPI_Datatype *filetype, char *datarep)	18
<pre>int MPI_File_iread(MPI_File fh, void *buf, int count,</pre>	19 20
MPI_Datatype datatype, MPI_Request *request)	21
<pre>int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,</pre>	22
MPI_Datatype datatype, MPI_Request *request)	23 24
int MPI_File_iread_shared(MPI_File fh, void *buf, int count,	25
MPI_Datatype datatype, MPI_Request *request)	26
<pre>int MPI_File_iwrite(MPI_File fh, const void *buf, int count,</pre>	27 28
	29
<pre>int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>	30
	31 32
<pre>int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	33
int MPI_File_open(MPI_Comm comm, const char *filename, int amode,	34
MPI_Info info, MPI_File *fh)	35 36
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)	37
	38
<pre>int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,</pre>	39 40
int MPI_File_read_all(MPI_File fh, void *buf, int count,	40 41
MPI_Datatype datatype, MPI_Status *status)	42
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,	43
MPI_Datatype datatype)	44 45
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	46
	47 48
	-40

1 2	int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,</pre>
3 4 5	int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>
6 7	int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>
8 9	int	<pre>MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>
10 11	int	MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
12 13 14	int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>
15	int	<pre>MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)</pre>
16 17 18	int	MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
19	int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
20 21	int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
22	int	MPI_File_set_atomicity(MPI_File fh, int flag)
23 24	int	MPI_File_set_info(MPI_File fh, MPI_Info info)
25 26	int	MPI_File_set_size(MPI_File fh, MPI_Offset size)
27 28	int	<pre>MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)</pre>
29 30	int	MPI_File_sync(MPI_File fh)
31 32	int	<pre>MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
33 34 35	int	<pre>MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
36 37	int	<pre>MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>
38 39 40	int	<pre>MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
41 42	int	<pre>MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
43 44 45	int	<pre>MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
46 47 48	int	<pre>MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype)</pre>

int	<pre>MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>	1 2
int	MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	3 4 5
int	<pre>MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>	6 7
int	<pre>MPI_File_write_ordered_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>	8 9 10
int	<pre>MPI_File_write_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	11 12
int	<pre>MPI_Register_datarep(const char *datarep,</pre>	13 14 15 16 17 18 19
A.2.	12 Language Bindings C Bindings	20 21
int	MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)	22
int	MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)	23 24
int	MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)	25
int	MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)	26
int	MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)	27 28
int	MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)	29 30
MPI_	_Fint MPI_Comm_c2f(MPI_Comm comm)	31
MPI_	_Comm MPI_Comm_f2c(MPI_Fint comm)	32 33
	_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	34
		35
	_Fint MPI_File_c2f(MPI_File file)	36 37
		38
	_File MPI_File_f2c(MPI_Fint file)	39 40
MPI_	_Fint MPI_Group_c2f(MPI_Group group)	40
MPI_	_Group MPI_Group_f2c(MPI_Fint group)	42
MPI_	_Fint MPI_Info_c2f(MPI_Info info)	43 44
MPI_	_Info MPI_Info_f2c(MPI_Fint info)	44 45
MPI	_Fint MPI_Message_c2f(MPI_Message message)	46
	_Message MPI_Message_f2c(MPI_Fint message)	47 48
т.п. т -	THESPARE IN T_HESPARE_ITC(IN I_LING MESPARE)	-10

1	MPI_Fint MPI_Op_c2f(MPI_Op op)
2	
3	MPI_Op MPI_Op_f2c(MPI_Fint op)
4 5	MPI_Fint MPI_Request_c2f(MPI_Request request)
6	MPI_Request MPI_Request_f2c(MPI_Fint request)
7 8	<pre>int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)</pre>
9 10	int MPI_Status_c2f08(const MPI_Status *c_status, MPI_F08_status *f08_status)
11 12	int MPI_Status_f082c(const MPI_F08_status *f08_status, MPI_Status *c_status)
13 14	int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
15 16	MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
17	MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
18 19	MPI_Fint MPI_Win_c2f(MPI_Win win)
20	MPI_Win MPI_Win_f2c(MPI_Fint win)
21	
22 23	A.2.13 Tools / Profiling Interface C Bindings
24	<pre>int MPI_Pcontrol(const int level,)</pre>
25 26	
27	A.2.14 Tools / MPI Tool Information Interface C Bindings
28 29	<pre>int MPI_T_category_changed(int *stamp)</pre>
30	<pre>int MPI_T_category_get_categories(int cat_index, int len, int indices[])</pre>
31 32	<pre>int MPI_T_category_get_cvars(int cat_index, int len, int indices[])</pre>
33 34 35	<pre>int MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>
36 37	<pre>int MPI_T_category_get_num(int *num_cat)</pre>
38	<pre>int MPI_T_category_get_pvars(int cat_index, int len, int indices[])</pre>
39 40	int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len, int
41	<pre>*verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind, int *scope)</pre>
42 43	int MPI_T_cvar_get_num(int *num_cvar)
44	
45	<pre>int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,</pre>
46 47	<pre>int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)</pre>
48	THE HETTER AT THE CHETTER (HETTER AT THE HEHRTE)

int	<pre>MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf)</pre>	1
int	MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf)	2 3
int	<pre>MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name, int *name_len)</pre>	4 5
int	<pre>MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value, char     *name, int *name_len)</pre>	6 7 8
int	MPI_T_finalize(void)	9
int	MPI_T_init_thread(int required, int *provided)	10 11
	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>	12
1110	<pre>int readonly, int *continuous, int *atomic)</pre>	13 14 15 16
int	MPI_T_pvar_get_num(int *num_pvar)	17
int	<pre>MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index, void *obj_handle, MPI_T_pvar_handle *handle, int *count)</pre>	18 19 20
int	<pre>MPI_T_pvar_handle_free(MPI_T_pvar_session session, MPI_T_pvar_handle</pre>	21 22
int	<pre>MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle, void* buf)</pre>	23 24 25
int	<pre>MPI_T_pvar_readreset(MPI_T_pvar_session session, MPI_T_pvar_handle handle, void* buf)</pre>	26 27
int	<pre>MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>	28 29
int	<pre>MPI_T_pvar_session_create(MPI_T_pvar_session *session)</pre>	30
int	MPI_T_pvar_session_free(MPI_T_pvar_session *session)	31 32
int	MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)	33
int	MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)	34 35
int	<pre>MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>	36 37 38
A.2.	15 Deprecated C Bindings	39 40
int	MPI_Attr_delete(MPI_Comm comm, int keyval)	41
int	MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)	42 43
int	MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)	44
	MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,	$45 \\ 46$
1110	void *attribute_val_in, void *attribute_val_out, int *flag)	40 47 48

int MPI\_Keyval\_create(MPI\_Copy\_function \*copy\_fn, MPI\_Delete\_function  $\mathbf{2}$ \*delete\_fn, int \*keyval, void\* extra\_state) int MPI\_Keyval\_free(int \*keyval)  $\mathbf{5}$ int MPI\_NULL\_COPY\_FN(MPI\_Comm oldcomm, int keyval, void \*extra\_state, void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag)  $\overline{7}$ int MPI\_NULL\_DELETE\_FN(MPI\_Comm comm, int keyval, void \*attribute\_val, void \*extra\_state)  $^{22}$  $^{24}$ 

	$_{2}^{1}$ ticket247-S.
A.3 Fortran 2008 Bindings with the mpi_f08 Module	3 4
A.3.1 Point-to-Point Communication Fortran 2008 Bindings	5
<pre>MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)     TYPE(*), DIMENSION(), INTENT(IN) :: buf</pre>	7 8
INTEGER, INTENT(IN) :: count, dest, tag	9
TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm	10 11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12 13
<pre>MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror) </pre>	13
BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	15 16
INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype	17
TYPE(MPI_Comm), INTENT(IN) :: comm	18 19
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_Buffer_attach(buffer, size, ierror) BIND(C)	21 22
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer	23
INTEGER, INTENT(IN) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24 25
MPI_Buffer_detach(buffer_addr, size, ierror) BIND(C)	26
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	27 28
TYPE(C_PTR), INTENT(OUT) :: buffer_addr INTEGER, INTENT(OUT) :: size	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30 31
MPI_Cancel(request, ierror) BIND(C)	32 33
TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Get_count(status, datatype, count, ierror) BIND(C)	35 36
TYPE(MPI_Status), INTENT(IN) :: status TYPE(MPI_Datatype), INTENT(IN) :: datatype	37
INTEGER, INTENT(OUT) :: count	38 39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
<pre>MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf</pre>	41 42
INTEGER, INTENT(IN) :: count, dest, tag	43
TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm	44 45
TYPE(MPI_Request), INTENT(OUT) :: request	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47 48

```
1
    MPI_Improbe(source, tag, comm, flag, message, status, ierror) BIND(C)
\mathbf{2}
         INTEGER, INTENT(IN) :: source, tag
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
8
    MPI_Imrecv(buf, count, datatype, message, request, ierror) BIND(C)
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
10
         INTEGER, INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Message), INTENT(INOUT) :: message
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
    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
22
    MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND(C)
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
24
         INTEGER, INTENT(IN) :: count, source, 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
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
37
    MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
38
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
39
         INTEGER, INTENT(IN) :: count, dest, tag
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
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
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
                                                                                 5
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
MPI_Mrecv(buf, count, datatype, message, status, ierror) BIND(C)
                                                                                12
                                                                                13
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
                                                                                14
                                                                                15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
                                                                                16
                                                                                17
    TYPE(MPI_Status) :: status
                                                                                18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                19
MPI_Probe(source, tag, comm, status, ierror) BIND(C)
                                                                                20
    INTEGER, INTENT(IN) :: source, tag
                                                                                21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                22
    TYPE(MPI_Status) :: status
                                                                                23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                24
                                                                                25
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C)
                                                                                26
    TYPE(*), DIMENSION(..) :: buf
                                                                                27
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                30
    TYPE(MPI_Status) :: status
                                                                                31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                32
MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
                                                                                33
             BIND(C)
                                                                                34
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                35
    INTEGER, INTENT(IN) :: count, source, 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
                                                                                41
MPI_Request_free(request, ierror) BIND(C)
                                                                                42
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                44
MPI_Request_get_status(request, flag, status, ierror) BIND(C)
                                                                                45
    TYPE(MPI_Request), INTENT(IN) :: request
                                                                                46
    LOGICAL, INTENT(OUT) :: flag
                                                                                 47
    TYPE(MPI_Status) :: status
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
    MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
3
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
4
         INTEGER, INTENT(IN) :: count, dest, tag
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_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
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
18
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
19
         INTEGER, INTENT(IN) :: count, dest, tag
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
    MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
25
                  BIND(C)
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
32
    MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
33
                  recvcount, recvtype, source, recvtag, comm, status, ierror)
34
                  BIND(C)
35
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
36
         TYPE(*), DIMENSION(..) :: recvbuf
37
         INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
38
         recvtag
39
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         TYPE(MPI_Status) :: status
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
    MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
45
                  comm, status, ierror) BIND(C)
46
         TYPE(*), DIMENSION(..) :: buf
47
         INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
48
```

TYPE(MPI_Datatype), INTENT(IN) :: datatype	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2
TYPE(MPI_Status) :: status	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)	5
TYPE(*), DIMENSION(), INTENT(IN) :: buf	6
INTEGER, INTENT(IN) :: count, dest, tag	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8 9
TYPE(MPI_Comm), INTENT(IN) :: comm	9 10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)	12
BIND(C)	14
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	15
INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype	16
TYPE(MPI_Datatype), INTENT(IN) datatype TYPE(MPI_Comm), INTENT(IN) :: comm	17
TYPE(MPI_COMMI), INTENT(IN) COMMI TYPE(MPI_Request), INTENT(OUT) :: request	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
INIEGER, OFFICIARE, INTENT(UCT) TETTOT	20
MPI_Start(request, ierror) BIND(C)	21
TYPE(MPI_Request), INTENT(INOUT) :: request	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
MPI_Startall(count, array_of_requests, ierror) BIND(C)	24
INTEGER, INTENT(IN) :: count	25
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	28
MPI_Test(request, flag, status, ierror) BIND(C)	29
TYPE(MPI_Request), INTENT(INOUT) :: request	30
LOGICAL, INTENT(OUT) :: flag TYPE(MPI_Status) :: status	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
INTEGER, OFFICIARE, INTENT(OUT) TETTOT	33
<pre>MPI_Test_cancelled(status, flag, ierror) BIND(C)</pre>	34
TYPE(MPI_Status), INTENT(IN) :: status	35
LOGICAL, INTENT(OUT) :: flag	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37 38
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)	39
BIND(C)	40
INTEGER, INTENT(IN) :: count	41
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)	42
LOGICAL, INTENT(OUT) :: flag	43
TYPE(MPI_Status) :: array_of_statuses(*)	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
MDI Togtony (count orrow of requests index flow status israe) DIND(()	46
<pre>MPI_Testany(count, array_of_requests, index, flag, status, ierror) BIND(C) INTEGER, INTENT(IN) :: count</pre>	47
INIEGER, INTENT(IN) COULD	48

```
1
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
2
         INTEGER, INTENT(OUT) :: index
3
         LOGICAL, INTENT(OUT) :: flag
4
         TYPE(MPI_Status) :: status
\mathbf{5}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
7
                  array_of_statuses, ierror) BIND(C)
8
         INTEGER, INTENT(IN) :: incount
9
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
10
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
11
         TYPE(MPI_Status) :: array_of_statuses(*)
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Wait(request, status, ierror) BIND(C)
15
         TYPE(MPI_Request), INTENT(INOUT) :: request
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) BIND(C)
19
         INTEGER, INTENT(IN) :: count
20
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
21
         TYPE(MPI_Status) :: array_of_statuses(*)
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
^{24}
     MPI_Waitany(count, array_of_requests, index, status, ierror) BIND(C)
25
         INTEGER, INTENT(IN) :: count
26
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
27
         INTEGER, INTENT(OUT) :: index
28
         TYPE(MPI_Status) :: status
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
^{31}
                  array_of_statuses, ierror) BIND(C)
32
         INTEGER, INTENT(IN) :: incount
33
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
34
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
35
         TYPE(MPI_Status) :: array_of_statuses(*)
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
39
     A.3.2 Datatypes Fortran 2008 Bindings
40
    MPI_Get_address(location, address, ierror) BIND(C)
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: location
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
    MPI_Get_elements(status, datatype, count, ierror) BIND(C)
46
         TYPE(MPI_Status), INTENT(IN) :: status
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    INTEGER, INTENT(OUT) :: count
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 3
MPI_Get_elements_x(status, datatype, count, ierror) BIND(C)
                                                                                 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
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
                                                                                 9
                                                                                 10
             BIND(C)
                                                                                 11
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                12
                                                                                13
    INTEGER, INTENT(IN) :: incount, outsize
                                                                                14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                15
    INTEGER, INTENT(INOUT) :: position
                                                                                 16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                18
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
                                                                                19
             position, ierror) BIND(C)
                                                                                20
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                21
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
                                                                                22
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                23
    INTEGER, INTENT(IN) :: incount
                                                                                24
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                25
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
                                                                                26
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
                                                                                27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                28
                                                                                29
MPI_Pack_external_size(datarep, incount, datatype, size, ierror) BIND(C)
                                                                                30
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                31
    INTEGER, INTENT(IN) :: incount
                                                                                32
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                33
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
                                                                                34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                35
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
                                                                                42
MPI_Type_commit(datatype, ierror) BIND(C)
                                                                                43
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
                                                                                44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                45
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
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     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
12
13
     MPI_Type_create_hindexed(count, array_of_blocklengths,
14
                  array_of_displacements, oldtype, newtype, ierror) BIND(C)
15
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
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
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
22
                  oldtype, newtype, ierror) BIND(C)
23
         INTEGER, INTENT(IN) :: count, blocklength
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
25
         array_of_displacements(count)
26
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
27
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
^{31}
                  ierror) BIND(C)
32
         INTEGER, INTENT(IN) :: count, blocklength
33
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
34
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
35
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
38
                  oldtype, newtype, ierror) BIND(C)
39
         INTEGER, INTENT(IN) :: count, blocklength,
40
         array_of_displacements(count)
41
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
42
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) BIND(C)
46
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
47
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
48
```

```
1
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 3
MPI_Type_create_struct(count, array_of_blocklengths,
                                                                                 4
             array_of_displacements, array_of_types, newtype, ierror)
                                                                                 5
             BIND(C)
                                                                                 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
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                 13
                                                                                 14
             array_of_starts, order, oldtype, newtype, ierror) BIND(C)
                                                                                 15
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
                                                                                 16
    array_of_subsizes(ndims), array_of_starts(ndims), order
                                                                                 17
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                 18
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 19
                                                                                 20
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
                                                                                 25
MPI_Type_free(datatype, ierror) BIND(C)
                                                                                 26
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
                                                                                 27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 28
MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
                                                                                 29
             array_of_integers, array_of_addresses, array_of_datatypes,
                                                                                 30
              ierror) BIND(C)
                                                                                 31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 32
    INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
                                                                                 33
    INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
                                                                                 34
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
                                                                                 35
    array_of_addresses(max_addresses)
                                                                                 36
    TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
                                                                                 37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 38
                                                                                 39
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
                                                                                 40
             combiner, ierror) BIND(C)
                                                                                 41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 42
    INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
                                                                                 43
    combiner
                                                                                 44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 45
MPI_Type_get_extent(datatype, lb, extent, ierror) BIND(C)
                                                                                 46
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 47
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Type_get_extent_x(datatype, lb, extent, ierror) BIND(C)
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) BIND(C)
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
    MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror) BIND(C)
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
    MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
17
                  oldtype, newtype, ierror) BIND(C)
18
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
19
         array_of_displacements(count)
20
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
21
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
22
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
23
     MPI_Type_size(datatype, size, ierror) BIND(C)
24
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
         INTEGER, INTENT(OUT) :: size
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Type_size_x(datatype, size, ierror) BIND(C)
29
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
33
                  BIND(C)
34
         INTEGER, INTENT(IN) :: count, blocklength, stride
35
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
36
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
40
                  ierror) BIND(C)
41
         TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
42
         TYPE(*), DIMENSION(..) :: outbuf
43
         INTEGER, INTENT(IN) :: insize, outcount
44
         INTEGER, INTENT(INOUT) :: position
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
                                                                                 2
             datatype, ierror) BIND(C)
                                                                                 3
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                 4
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                 5
                                                                                 6
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
                                                                                 7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
                                                                                 8
    INTEGER, INTENT(IN) :: outcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 9
                                                                                 10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 11
                                                                                12
A.3.3 Collective Communication Fortran 2008 Bindings
                                                                                13
                                                                                14
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                15
             comm, ierror) BIND(C)
                                                                                 16
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 17
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 18
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                19
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                22
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                23
             recvtype, comm, ierror) BIND(C)
                                                                                24
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                25
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                26
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                27
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                30
                                                                                31
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
                                                                                32
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                33
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                34
    INTEGER, INTENT(IN) :: count
                                                                                35
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                36
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                39
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                40
             comm, ierror) BIND(C)
                                                                                41
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                42
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                43
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 47
                                                                                 48
```

```
1
    MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
\mathbf{2}
                  rdispls, recvtype, comm, ierror) BIND(C)
3
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
4
         TYPE(*), DIMENSION(..) :: recvbuf
5
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
6
         rdispls(*)
7
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
11
                  rdispls, recvtypes, comm, ierror) BIND(C)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
13
         TYPE(*), DIMENSION(..) :: recvbuf
14
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
15
         rdispls(*)
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
17
         TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Barrier(comm, ierror) BIND(C)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Bcast(buffer, count, datatype, root, comm, ierror) BIND(C)
25
         TYPE(*), DIMENSION(...) :: buffer
26
         INTEGER, INTENT(IN) :: count, root
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
32
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
33
         TYPE(*), DIMENSION(..) :: recvbuf
34
         INTEGER, INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Op), INTENT(IN) :: op
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
40
                  root, comm, ierror) BIND(C)
41
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
42
         TYPE(*), DIMENSION(..) :: recvbuf
43
         INTEGER, INTENT(IN) :: sendcount, 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
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                 2
             recvtype, root, comm, ierror) BIND(C)
                                                                                 3
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 4
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
                                                                                 5
                                                                                 6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 7
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 8
                                                                                 9
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 10
             comm, request, ierror) BIND(C)
                                                                                11
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                12
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                13
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                14
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 16
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                18
                                                                                19
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, request, ierror) BIND(C)
                                                                                20
                                                                                21
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                22
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                23
    INTEGER, INTENT(IN) :: sendcount
                                                                                24
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                26
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                29
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
                                                                                30
             ierror) BIND(C)
                                                                                31
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 32
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                33
    INTEGER, INTENT(IN) :: count
                                                                                34
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                35
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                39
                                                                                40
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                41
             comm, request, ierror) BIND(C)
                                                                                42
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                43
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                44
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 45
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 47
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
3
                  rdispls, recvtype, comm, request, ierror) BIND(C)
4
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
7
         recvcounts(*), rdispls(*)
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
14
                  recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
15
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
16
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
17
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
18
         recvcounts(*), rdispls(*)
19
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
20
         recvtypes(*)
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Request), INTENT(OUT) :: request
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Ibarrier(comm, request, ierror) BIND(C)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C)
30
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
31
         INTEGER, INTENT(IN) :: count, root
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
37
                  BIND(C)
38
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
40
         INTEGER, INTENT(IN) :: count
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_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
48
```

1 root, comm, request, ierror) BIND(C) TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf  $\mathbf{2}$ 3 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendcount, recvcount, root 4 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 56 TYPE(MPI\_Comm), INTENT(IN) :: comm 7 TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI\_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 10 recvtype, root, comm, request, ierror) BIND(C) 11 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 12TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 13 INTEGER, INTENT(IN) :: sendcount, root 14INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(\*), displs(\*) 15TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 16 TYPE(MPI\_Comm), INTENT(IN) :: comm 17TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI\_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request, 2021ierror) BIND(C) 22 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 23TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 24INTEGER, INTENT(IN) :: count, root 25TYPE(MPI\_Datatype), INTENT(IN) :: datatype 26TYPE(MPI\_Op), INTENT(IN) :: op 27TYPE(MPI\_Comm), INTENT(IN) :: comm 28 TYPE(MPI\_Request), INTENT(OUT) :: request 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI\_Ireduce\_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, 31request, ierror) BIND(C) 32 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 33 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 34 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 35 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI\_Op), INTENT(IN) :: op 37 TYPE(MPI\_Comm), INTENT(IN) :: comm 38 TYPE(MPI\_Request), INTENT(OUT) :: request 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4041 MPI\_Ireduce\_scatter\_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 42request, ierror) BIND(C) 43 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 44TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 45INTEGER, INTENT(IN) :: recvcount 46TYPE(MPI\_Datatype), INTENT(IN) :: datatype 47TYPE(MPI\_Op), INTENT(IN) :: op 48

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
5
                  BIND(C)
6
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER, INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Op), INTENT(IN) :: op
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
16
                  root, comm, request, ierror) BIND(C)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
19
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Request), INTENT(OUT) :: request
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
25
                  recvtype, root, comm, request, ierror) BIND(C)
26
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
27
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
28
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
29
         INTEGER, INTENT(IN) :: recvcount, root
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
34
35
     MPI_Op_commutative(op, commute, ierror) BIND(C)
36
         TYPE(MPI_Op), INTENT(IN) :: op
37
         LOGICAL, INTENT(OUT) :: commute
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Op_create(user_fn, commute, op, ierror) BIND(C)
40
         PROCEDURE(MPI_User_function) :: user_fn
41
         LOGICAL, INTENT(IN) :: commute
42
         TYPE(MPI_Op), INTENT(OUT) :: op
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
    MPI_Op_free(op, ierror) BIND(C)
46
         TYPE(MPI_Op), INTENT(INOUT) :: op
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
                                                                                 2
             BIND(C)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 4
    INTEGER, INTENT(IN) :: count, root
                                                                                 5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 6
                                                                                 7
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 9
                                                                                 10
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C)
                                                                                 11
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
                                                                                 12
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                 13
    INTEGER, INTENT(IN) :: count
                                                                                 14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 15
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 17
                                                                                 18
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                 19
             ierror) BIND(C)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 20
                                                                                 21
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 22
    INTEGER, INTENT(IN) :: recvcounts(*)
                                                                                 23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 24
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 27
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                 28
             ierror) BIND(C)
                                                                                 29
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 30
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 31
    INTEGER, INTENT(IN) :: recvcount
                                                                                 32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 33
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 36
                                                                                 37
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
                                                                                 38
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 39
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 40
    INTEGER, INTENT(IN) :: count
                                                                                 41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 42
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 45
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 46
             root, comm, ierror) BIND(C)
                                                                                 47
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 48
```

```
1
         TYPE(*), DIMENSION(..) :: recvbuf
\mathbf{2}
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
3
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
7
                  recvtype, root, comm, ierror) BIND(C)
8
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
9
         TYPE(*), DIMENSION(..) :: recvbuf
10
         INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
11
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
16
     A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
17
     MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
18
                  attribute_val_out, flag, ierror) BIND(C)
19
         TYPE(MPI_Comm), INTENT(IN) :: oldcomm
20
         INTEGER, INTENT(IN) :: comm_keyval
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
22
         attribute_val_in
23
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
24
         LOGICAL, INTENT(OUT) :: flag
25
         INTEGER, INTENT(OUT) :: ierror
26
27
     MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
28
                  attribute_val_out, flag, ierror) BIND(C)
29
         TYPE(MPI_Comm), INTENT(IN) :: oldcomm
30
         INTEGER, INTENT(IN) :: comm_keyval
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
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
     MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
37
                  ierror) BIND(C)
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, INTENT(IN) :: comm_keyval
40
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
41
         extra_state
42
         INTEGER, INTENT(OUT) :: ierror
43
^{44}
     MPI_Comm_compare(comm1, comm2, result, ierror) BIND(C)
45
         TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
46
         INTEGER, INTENT(OUT) :: result
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

<pre>MPI_Comm_create(comm, group, newcomm, ierror) BIND(C)</pre>	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2 3
TYPE(MPI_Group), INTENT(IN) :: group	3
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	4 5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)	7
TYPE(MPI_Comm), INTENT(IN) :: comm	8
TYPE(MPI_Group), INTENT(IN) :: group	9
INTEGER, INTENT(IN) :: tag	10
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
<pre>MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,</pre>	13 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	19
	20
MPI_Comm_delete_attr(comm, comm_keyval, ierror) BIND(C)	21
TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: comm_keyval	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
INTEGER, OFFICIARE, INTENT(COT) TETTOT	24
MPI_Comm_dup(comm, newcomm, ierror) BIND(C)	25
TYPE(MPI_Comm), INTENT(IN) :: comm	26 27
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_Comm_dup_with_info(comm, info, newcomm, ierror) BIND(C)	30
TYPE(MPI_Comm), INTENT(IN) :: comm	31
TYPE(MPI_Info), INTENT(IN) :: info	32
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Comm_free(comm, ierror) BIND(C)	35
TYPE(MPI_Comm), INTENT(INOUT) :: comm	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
	38
MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)	39
INTEGER, INTENT(INOUT) :: comm_keyval	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
<pre>MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror) BIND(C)</pre>	42
TYPE(MPI_Comm), INTENT(IN) :: comm	43
INTEGER, INTENT(IN) :: comm_keyval	44 45
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val</pre>	45 46
LOGICAL, INTENT(OUT) :: flag	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
     MPI_Comm_get_info(comm, info_used, ierror) BIND(C)
\mathbf{2}
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         TYPE(MPI_Info), INTENT(OUT) :: info_used
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Comm_get_name(comm, comm_name, resultlen, ierror) BIND(C)
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
8
         INTEGER, INTENT(OUT) :: resultlen
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Comm_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
     MPI_Comm_idup(comm, newcomm, request, ierror) BIND(C)
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
18
         TYPE(MPI_Request), INTENT(OUT) :: request
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Comm_rank(comm, rank, ierror) BIND(C)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, INTENT(OUT) :: rank
^{24}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Comm_remote_group(comm, group, ierror) BIND(C)
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         TYPE(MPI_Group), INTENT(OUT) :: group
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Comm_remote_size(comm, size, ierror) BIND(C)
^{31}
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         INTEGER, INTENT(OUT) :: size
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) BIND(C)
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, INTENT(IN) :: comm_keyval
37
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Comm_set_info(MPI_Comm comm, MPI_Info info) BIND(C)
41
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
    MPI_Comm_set_name(comm, comm_name, ierror) BIND(C)
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         CHARACTER(LEN=*), INTENT(IN) :: comm_name
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_Comm_size(comm, size, ierror) BIND(C)	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2
INTEGER, INTENT(OUT) :: size	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Comm_split(comm, color, key, newcomm, ierror) BIND(C)	5 6
TYPE(MPI_Comm), INTENT(IN) :: comm	7
INTEGER, INTENT(IN) :: color, key	8
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) BIND(C)	11
TYPE(MPI_Comm), INTENT(IN) :: comm	12
INTEGER, INTENT(IN) :: split_type, key	13
TYPE(MPI_Info), INTENT(IN) :: info	14
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
<pre>MPI_Comm_test_inter(comm, flag, ierror) BIND(C)</pre>	17
TYPE(MPI_Comm), INTENT(IN) :: comm	18
LOGICAL, INTENT(OUT) :: flag	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20 21
	21
MPI_Group_compare(group1, group2, result, ierror) BIND(C)	22
TYPE(MPI_Group), INTENT(IN) :: group1, group2	24
INTEGER, INTENT(OUT) :: result	25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	26
<pre>MPI_Group_difference(group1, group2, newgroup, ierror) BIND(C)</pre>	27
TYPE(MPI_Group), INTENT(IN) :: group1, group2	28
TYPE(MPI_Group), INTENT(OUT) :: newgroup	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
<pre>MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C)</pre>	31
TYPE(MPI_Group), INTENT(IN) :: group	32
INTEGER, INTENT(IN) :: n, ranks(n)	33
TYPE(MPI_Group), INTENT(OUT) :: newgroup	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
<pre>MPI_Group_free(group, ierror) BIND(C)</pre>	36
TYPE(MPI_Group), INTENT(INOUT) :: group	37 38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
	40
<pre>MPI_Group_incl(group, n, ranks, newgroup, ierror) BIND(C)</pre>	41
TYPE(MPI_Group), INTENT(IN) :: group	42
INTEGER, INTENT(IN) :: n, ranks(n)	43
TYPE(MPI_Group), INTENT(OUT) :: newgroup	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
<pre>MPI_Group_intersection(group1, group2, newgroup, ierror) BIND(C)</pre>	46
TYPE(MPI_Group), INTENT(IN) :: group1, group2	47
TYPE(MPI_Group), INTENT(OUT) :: newgroup	48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Group_range_excl(group, n, ranges, newgroup, ierror) BIND(C)
3
         TYPE(MPI_Group), INTENT(IN) :: group
4
         INTEGER, INTENT(IN) :: n, ranges(3,n)
5
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)
9
         TYPE(MPI_Group), INTENT(IN) :: group
10
         INTEGER, INTENT(IN) :: n, ranges(3,n)
11
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Group_rank(group, rank, ierror) BIND(C)
14
         TYPE(MPI_Group), INTENT(IN) :: group
15
         INTEGER, INTENT(OUT) :: rank
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     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
     MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
23
                  BIND(C)
24
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
25
         INTEGER, INTENT(IN) :: n, ranks1(n)
26
         INTEGER, INTENT(OUT) :: ranks2(n)
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
    MPI_Group_union(group1, group2, newgroup, ierror) BIND(C)
30
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
^{31}
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
34
                  tag, newintercomm, ierror) BIND(C)
35
         TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
36
         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
37
         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) BIND(C)
41
         TYPE(MPI_Comm), INTENT(IN) :: intercomm
42
         LOGICAL, INTENT(IN) :: high
43
         TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
46
                  attribute_val_out, flag, ierror) BIND(C)
47
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
48
```

1 INTEGER, INTENT(IN) :: type\_keyval  $^{2}$ INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state, 3 attribute\_val\_in 4 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val\_out LOGICAL, INTENT(OUT) :: flag 56 INTEGER, INTENT(OUT) :: ierror 7 MPI\_TYPE\_NULL\_COPY\_FN(oldtype, type\_keyval, extra\_state, attribute\_val\_in, 8 attribute\_val\_out, flag, ierror) BIND(C) 9 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 10INTEGER, INTENT(IN) :: type\_keyval 11 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state, 12attribute\_val\_in 13 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val\_out 14LOGICAL, INTENT(OUT) :: flag 15INTEGER, INTENT(OUT) :: ierror 1617MPI\_TYPE\_NULL\_DELETE\_FN(datatype, type\_keyval, attribute\_val, extra\_state, 18 ierror) BIND(C) 19 TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: type\_keyval 2021INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: attribute\_val, 22 extra\_state 23INTEGER, INTENT(OUT) :: ierror 24MPI\_Type\_create\_keyval(type\_copy\_attr\_fn, type\_delete\_attr\_fn, type\_keyval, 25extra\_state, ierror) BIND(C) 26PROCEDURE(MPI\_Type\_copy\_attr\_function) :: type\_copy\_attr\_fn 27PROCEDURE(MPI\_Type\_delete\_attr\_function) :: type\_delete\_attr\_fn 28 INTEGER, INTENT(OUT) :: type\_keyval 29 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI\_Type\_delete\_attr(datatype, type\_keyval, ierror) BIND(C) 33 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 34 INTEGER, INTENT(IN) :: type\_keyval 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI\_Type\_free\_keyval(type\_keyval, ierror) BIND(C) 37 INTEGER, INTENT(INOUT) :: type\_keyval 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40MPI\_Type\_get\_attr(datatype, type\_keyval, attribute\_val, flag, ierror) 41 BIND(C) 42TYPE(MPI\_Datatype), INTENT(IN) :: datatype 43 INTEGER, INTENT(IN) :: type\_keyval 44INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val 45LOGICAL, INTENT(OUT) :: flag 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47MPI\_Type\_get\_name(datatype, type\_name, resultlen, ierror) BIND(C) 48

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
\mathbf{2}
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
3
         INTEGER, INTENT(OUT) :: resultlen
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)
6
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
         INTEGER, INTENT(IN) :: type_keyval
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
^{11}
    MPI_Type_set_name(datatype, type_name, ierror) BIND(C)
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         CHARACTER(LEN=*), INTENT(IN) :: type_name
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
16
                  attribute_val_out, flag, ierror) BIND(C)
17
         INTEGER, INTENT(IN) :: oldwin, win_keyval
18
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
19
         attribute_val_in
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
21
         LOGICAL, INTENT(OUT) :: flag
22
         INTEGER, INTENT(OUT) :: ierror
23
^{24}
     MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
25
                  attribute_val_out, flag, ierror) BIND(C)
26
         INTEGER, INTENT(IN) :: oldwin, win_keyval
27
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
28
         attribute_val_in
^{29}
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
30
         LOGICAL, INTENT(OUT) :: flag
31
         INTEGER, INTENT(OUT) :: ierror
32
     MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
33
                  BIND(C)
34
         TYPE(MPI_Win), INTENT(IN) :: win
35
         INTEGER, INTENT(IN) :: win_keyval
36
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
37
         extra_state
38
         INTEGER, INTENT(OUT) :: ierror
39
40
     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
41
                  extra_state, ierror) BIND(C)
42
         PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
43
         PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
44
         INTEGER, INTENT(OUT) :: win_keyval
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_Win_delete_attr(win, win_keyval, ierror) BIND(C)
48
```

TYPE(MPI_Win), INTENT(IN) :: win	1
INTEGER, INTENT(IN) :: win_keyval INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2 3
	4
MPI_Win_free_keyval(win_keyval, ierror) BIND(C)	5
INTEGER, INTENT(INOUT) :: win_keyval	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror) BIND(C)	8
TYPE(MPI_Win), INTENT(IN) :: win	9
INTEGER, INTENT(IN) :: win_keyval	10
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val</pre>	11
LOGICAL, INTENT(OUT) :: flag	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13 14
MPI_Win_get_name(win, win_name, resultlen, ierror) BIND(C)	14
TYPE(MPI_Win), INTENT(IN) :: win	16
CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name	17
INTEGER, INTENT(OUT) :: resultlen	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror) BIND(C)	20
TYPE(MPI_Win), INTENT(IN) :: win	21
INTEGER, INTENT(IN) :: win_keyval	22
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val</pre>	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
MPI_Win_set_name(win, win_name, ierror) BIND(C)	25 26
TYPE(MPI_Win), INTENT(IN) :: win	20 27
CHARACTER(LEN=*), INTENT(IN) :: win_name	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
	30
A 2 E Dracess Tenaleries Fortran 2008 Bindings	31
A.3.5 Process Topologies Fortran 2008 Bindings	32
<pre>MPI_Cart_coords(comm, rank, maxdims, coords, ierror) BIND(C)</pre>	33
TYPE(MPI_Comm), INTENT(IN) :: comm	34
INTEGER, INTENT(IN) :: rank, maxdims	35
INTEGER, INTENT(OUT) :: coords(maxdims)	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)	38
BIND(C)	39 40
TYPE(MPI_Comm), INTENT(IN) :: comm_old	40
<pre>INTEGER, INTENT(IN) :: ndims, dims(ndims)</pre>	42
LOGICAL, INTENT(IN) :: periods(ndims), reorder	43
TYPE(MPI_Comm), INTENT(OUT) :: comm_cart	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) BIND(C)	46
TYPE(MPI_Comm), INTENT(IN) :: comm	47
INTEGER, INTENT(IN) :: maxdims	48

```
1
         INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
\mathbf{2}
         LOGICAL, INTENT(OUT) :: periods(maxdims)
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) BIND(C)
5
         TYPE(MPI_Comm), INTENT(IN) :: comm
6
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
7
         LOGICAL, INTENT(IN) :: periods(ndims)
8
         INTEGER, INTENT(OUT) :: newrank
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Cart_rank(comm, coords, rank, ierror) BIND(C)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         INTEGER, INTENT(IN) :: coords(*)
14
         INTEGER, INTENT(OUT) :: rank
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
17
                  BIND(C)
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         INTEGER, INTENT(IN) :: direction, disp
20
         INTEGER, INTENT(OUT) :: rank_source, rank_dest
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Cart_sub(comm, remain_dims, newcomm, ierror) BIND(C)
^{24}
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         LOGICAL, INTENT(IN) :: remain_dims(*)
26
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Cartdim_get(comm, ndims, ierror) BIND(C)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, INTENT(OUT) :: ndims
^{31}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Dims_create(nnodes, ndims, dims, ierror) BIND(C)
34
         INTEGER, INTENT(IN) :: nnodes, ndims
35
         INTEGER, INTENT(INOUT) :: dims(ndims)
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
38
                  info, reorder, comm_dist_graph, ierror) BIND(C)
39
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
40
         INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
41
         INTEGER, INTENT(IN) :: weights(*)
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         LOGICAL, INTENT(IN) :: reorder
44
         TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
48
```

```
1
             outdegree, destinations, destweights, info, reorder,
                                                                                 2
             comm_dist_graph, ierror) BIND(C)
                                                                                 3
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
                                                                                 4
    destinations(outdegree)
                                                                                 5
                                                                                 6
    INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
                                                                                 7
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 8
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                 9
                                                                                 10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 11
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
                                                                                 12
             maxoutdegree, destinations, destweights, ierror) BIND(C)
                                                                                 13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 14
    INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
                                                                                 15
    INTEGER, INTENT(OUT) :: sources(maxindegree),
                                                                                 16
    destinations(maxoutdegree)
                                                                                 17
    INTEGER :: sourceweights(*), destweights(*)
                                                                                 18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 19
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
                                                                                 20
                                                                                 21
             BIND(C)
                                                                                 22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 23
    INTEGER, INTENT(OUT) :: indegree, outdegree
                                                                                 24
    LOGICAL, INTENT(OUT) :: weighted
                                                                                 25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 26
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
                                                                                 27
             ierror) BIND(C)
                                                                                 28
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                 29
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                 30
    LOGICAL, INTENT(IN) :: reorder
                                                                                 31
    TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
                                                                                 32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 33
                                                                                 34
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) BIND(C)
                                                                                 35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 36
    INTEGER, INTENT(IN) :: maxindex, maxedges
                                                                                 37
    INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
                                                                                 38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 39
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) BIND(C)
                                                                                 40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 41
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                 42
    INTEGER, INTENT(OUT) :: newrank
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
                                                                                 45
MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) BIND(C)
                                                                                 46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 47
    INTEGER, INTENT(IN) :: rank, maxneighbors
                                                                                 48
```

```
1
         INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) BIND(C)
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         INTEGER, INTENT(IN) :: rank
6
         INTEGER, INTENT(OUT) :: nneighbors
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Graphdims_get(comm, nnodes, nedges, ierror) BIND(C)
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         INTEGER, INTENT(OUT) :: nnodes, nedges
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
14
                  recvtype, comm, request, ierror) BIND(C)
15
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
16
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
17
         INTEGER, INTENT(IN) :: sendcount, recvcount
18
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
^{24}
                  displs, recvtype, comm, request, ierror) BIND(C)
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27
         INTEGER, INTENT(IN) :: sendcount
28
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
29
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         TYPE(MPI_Request), INTENT(OUT) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
34
                  recvtype, comm, request, ierror) BIND(C)
35
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
37
         INTEGER, INTENT(IN) :: sendcount, recvcount
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
42
43
    MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
44
                  recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)
45
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
48
```

```
recvcounts(*), rdispls(*)
                                                                                 1
                                                                                 2
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 3
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 5
                                                                                 6
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                 7
             recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
                                                                                 8
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 10
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                 11
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                 12
    sdispls(*), rdispls(*)
                                                                                 13
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                 14
    recvtypes(*)
                                                                                 15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 16
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 18
                                                                                 19
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror) BIND(C)
                                                                                 20
                                                                                 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_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                 28
             displs, recvtype, comm, ierror) BIND(C)
                                                                                 29
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 30
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 31
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                 32
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 35
                                                                                 36
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                 37
             recvtype, comm, ierror) BIND(C)
                                                                                 38
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 39
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 40
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 41
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                 45
             recvcounts, rdispls, recvtype, comm, ierror) BIND(C)
                                                                                 46
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 47
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 48
```

```
1
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
2
         rdispls(*)
3
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
\mathbf{5}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
7
                  recvcounts, rdispls, recvtypes, comm, ierror) BIND(C)
8
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
9
         TYPE(*), DIMENSION(..) :: recvbuf
10
         INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
11
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
12
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Topo_test(comm, status, ierror) BIND(C)
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, INTENT(OUT) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     A.3.6 MPI Environmental Management Fortran 2008 Bindings
22
23
     DOUBLE PRECISION MPI_Wtick() BIND(C)
24
    DOUBLE PRECISION MPI_Wtime() BIND(C)
25
26
     MPI_Abort(comm, errorcode, ierror) BIND(C)
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         INTEGER, INTENT(IN) :: errorcode
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Add_error_class(errorclass, ierror) BIND(C)
^{31}
         INTEGER, INTENT(OUT) :: errorclass
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Add_error_code(errorclass, errorcode, ierror) BIND(C)
35
         INTEGER, INTENT(IN) :: errorclass
36
         INTEGER, INTENT(OUT) :: errorcode
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Add_error_string(errorcode, string, ierror) BIND(C)
39
         INTEGER, INTENT(IN) :: errorcode
40
41
         CHARACTER(LEN=*), INTENT(IN) :: string
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
44
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
46
         TYPE(MPI_Info), INTENT(IN) :: info
47
         TYPE(C_PTR), INTENT(OUT) :: baseptr
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Comm_call_errhandler(comm, errorcode, ierror) BIND(C)	2
TYPE(MPI_Comm), INTENT(IN) :: comm	$\frac{3}{4}$
INTEGER, INTENT(IN) :: errorcode	5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) BIND(C)	7
PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn	8
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_Comm_get_errhandler(comm, errhandler, ierror) BIND(C)	12
TYPE(MPI_Comm), INTENT(IN) :: comm	13
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
INTEGER, OFITONAL, INTENI(OOI) TEITOT	15
<pre>MPI_Comm_set_errhandler(comm, errhandler, ierror) BIND(C)</pre>	16
TYPE(MPI_Comm), INTENT(IN) :: comm	17 18
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_Errhandler_free(errhandler, ierror) BIND(C)	21
TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
MPI_Error_class(errorcode, errorclass, ierror) BIND(C)	24
INTEGER, INTENT(IN) :: errorcode	25
INTEGER, INTENT(OUT) :: errorclass	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	28
MPI_Error_string(errorcode, string, resultlen, ierror) BIND(C)	29
INTEGER, INTENT(IN) :: errorcode CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string	30
INTEGER, INTENT(OUT) :: resultlen	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 33
	34
MPI_File_call_errhandler(fh, errorcode, ierror) BIND(C)	35
TYPE(MPI_File), INTENT(IN) :: fh	36
INTEGER, INTENT(IN) :: errorcode	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
<pre>MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C)</pre>	39
<pre>PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn</pre>	40
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)	43
TYPE(MPI_File), INTENT(IN) :: file	44
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46 47
MPI_File_set_errhandler(file, errhandler, ierror) BIND(C)	47
	10

```
1
         TYPE(MPI_File), INTENT(IN) :: file
\mathbf{2}
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Finalize(ierror) BIND(C)
\mathbf{5}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Finalized(flag, ierror) BIND(C)
8
         LOGICAL, INTENT(OUT) :: flag
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_Free_mem(base, ierror) BIND(C)
11
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Get_library_version(version, resulten, ierror) BIND(C)
15
         CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
16
         INTEGER, INTENT(OUT) :: resultlen
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
    MPI_Get_processor_name(name, resultlen, ierror) BIND(C)
19
         CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
20
         INTEGER, INTENT(OUT) :: resultlen
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Get_version(version, subversion, ierror) BIND(C)
^{24}
         INTEGER, INTENT(OUT) :: version, subversion
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
    MPI_Init(ierror) BIND(C)
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Initialized(flag, ierror) BIND(C)
30
         LOGICAL, INTENT(OUT) :: flag
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Win_call_errhandler(win, errorcode, ierror) BIND(C)
33
         TYPE(MPI_Win), INTENT(IN) :: win
34
         INTEGER, INTENT(IN) :: errorcode
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror) BIND(C)
38
         PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
39
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
    MPI_Win_get_errhandler(win, errhandler, ierror) BIND(C)
42
         TYPE(MPI_Win), INTENT(IN) :: win
43
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     MPI_Win_set_errhandler(win, errhandler, ierror) BIND(C)
47
         TYPE(MPI_Win), INTENT(IN) :: win
48
```

TYPE(MPI_Errhandler), INTENT(IN) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
INTEGER, OFITONAL, INTENT(001) TETTOT	3
A.3.7 The Info Object Fortran 2008 Bindings	4 5
MPI_Info_create(info, ierror) BIND(C)	6
TYPE(MPI_Info), INTENT(OUT) :: info	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8 9
MPI_Info_delete(info, key, ierror) BIND(C)	9 10
TYPE(MPI_Info), INTENT(IN) :: info	11
CHARACTER(LEN=*), INTENT(IN) :: key	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Info_dup(info, newinfo, ierror) BIND(C)	14 15
TYPE(MPI_Info), INTENT(IN) :: info	16
TYPE(MPI_Info), INTENT(OUT) :: newinfo INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
	18
<pre>MPI_Info_free(info, ierror) BIND(C)     TYPE(MPI_Info), INTENT(INOUT) :: info</pre>	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20 21
	22
<pre>MPI_Info_get(info, key, valuelen, value, flag, ierror) BIND(C)         TYPE(MPI_Info), INTENT(IN) :: info</pre>	23
CHARACTER(LEN=*), INTENT(IN) :: key	24
INTEGER, INTENT(IN) :: valuelen	25
CHARACTER(LEN=valuelen), INTENT(OUT) :: value	26 27
LOGICAL, INTENT(OUT) :: flag	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
MPI_Info_get_nkeys(info, nkeys, ierror) BIND(C)	30
TYPE(MPI_Info), INTENT(IN) :: info	31
INTEGER, INTENT(OUT) :: nkeys INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 33
	34
<pre>MPI_Info_get_nthkey(info, n, key, ierror) BIND(C)     TYPE(MPI_Info), INTENT(IN) :: info</pre>	35
INTEGER, INTENT(IN) :: n	36
CHARACTER(LEN=*), INTENT(OUT) :: key	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38 39
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) BIND(C)	40
TYPE(MPI_Info), INTENT(IN) :: info	41
CHARACTER(LEN=*), INTENT(IN) :: key	42
INTEGER, INTENT(OUT) :: valuelen	43
LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44 45
	45 46
MPI_Info_set(info, key, value, ierror) BIND(C)	47
TYPE(MPI_Info), INTENT(IN) :: info	48

```
1
         CHARACTER(LEN=*), INTENT(IN) :: key, value
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
4
     A.3.8 Process Creation and Management Fortran 2008 Bindings
5
6
    MPI_Close_port(port_name, ierror) BIND(C)
7
         CHARACTER(LEN=*), INTENT(IN) :: port_name
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) BIND(C)
10
         CHARACTER(LEN=*), INTENT(IN) :: port_name
11
         TYPE(MPI_Info), INTENT(IN) :: info
12
         INTEGER, INTENT(IN) :: root
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror) BIND(C)
18
         CHARACTER(LEN=*), INTENT(IN) :: port_name
19
         TYPE(MPI_Info), INTENT(IN) :: info
20
         INTEGER, INTENT(IN) :: root
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Comm_disconnect(comm, ierror) BIND(C)
25
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Comm_get_parent(parent, ierror) BIND(C)
29
         TYPE(MPI_Comm), INTENT(OUT) :: parent
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     MPI_Comm_join(fd, intercomm, ierror) BIND(C)
32
         INTEGER, INTENT(IN) :: fd
33
34
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
37
                   array_of_errcodes, ierror) BIND(C)
38
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
39
         INTEGER, INTENT(IN) :: maxprocs, root
40
         TYPE(MPI_Info), INTENT(IN) :: info
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
43
         INTEGER :: array_of_errcodes(*)
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
46
47
                  array_of_maxprocs, array_of_info, root, comm, intercomm,
48
                  array_of_errcodes, ierror) BIND(C)
```

1 INTEGER, INTENT(IN) :: count, array\_of\_maxprocs(\*), root 2 CHARACTER(LEN=\*), INTENT(IN) :: array\_of\_commands(\*) 3 CHARACTER(LEN=\*), INTENT(IN) :: array\_of\_argv(count, \*) 4 TYPE(MPI\_Info), INTENT(IN) :: array\_of\_info(\*) TYPE(MPI\_Comm), INTENT(IN) :: comm 5TYPE(MPI\_Comm), INTENT(OUT) :: intercomm 6 7 INTEGER :: array\_of\_errcodes(\*) INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI\_Lookup\_name(service\_name, info, port\_name, ierror) BIND(C) 10CHARACTER(LEN=\*), INTENT(IN) :: service\_name 11 TYPE(MPI\_Info), INTENT(IN) :: info 12CHARACTER(LEN=MPI\_MAX\_PORT\_NAME), INTENT(OUT) :: port\_name 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_Open\_port(info, port\_name, ierror) BIND(C) 16TYPE(MPI\_Info), INTENT(IN) :: info 17CHARACTER(LEN=MPI\_MAX\_PORT\_NAME), INTENT(OUT) :: port\_name 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI\_Publish\_name(service\_name, info, port\_name, ierror) BIND(C) 20TYPE(MPI\_Info), INTENT(IN) :: info 21CHARACTER(LEN=\*), INTENT(IN) :: service\_name, port\_name 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2324MPI\_Unpublish\_name(service\_name, info, port\_name, ierror) BIND(C) 25CHARACTER(LEN=\*), INTENT(IN) :: service\_name, port\_name 26TYPE(MPI\_Info), INTENT(IN) :: info 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28 29A.3.9 One-Sided Communications Fortran 2008 Bindings 30 31MPI\_Accumulate(origin\_addr, origin\_count, origin\_datatype, target\_rank, 32 target\_disp, target\_count, target\_datatype, op, win, ierror) 33 BIND(C) 34 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 35INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 36 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 37 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 38 TYPE(MPI\_Op), INTENT(IN) :: op 39 TYPE(MPI\_Win), INTENT(IN) :: win 40INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI\_Compare\_and\_swap(origin\_addr, compare\_addr, result\_addr, datatype, 42target\_rank, target\_disp, win, ierror) BIND(C) 43 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 44TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: compare\_addr 45TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 46TYPE(MPI\_Datatype), INTENT(IN) :: datatype 47INTEGER, INTENT(IN) :: target\_rank 48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
2
         TYPE(MPI_Win), INTENT(IN) :: win
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
5
                  target_disp, op, win, ierror) BIND(C)
6
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER, INTENT(IN) :: target_rank
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
11
         TYPE(MPI_Op), INTENT(IN) :: op
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
14
15
    MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
16
                  target_disp, target_count, target_datatype, win, ierror)
17
                  BIND(C)
18
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
19
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
20
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
22
         TYPE(MPI_Win), INTENT(IN) :: win
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
25
                  result_count, result_datatype, target_rank, target_disp,
26
                  target_count, target_datatype, op, win, ierror) BIND(C)
27
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
28
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
29
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
30
         target_count
31
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
32
         result_datatype
33
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
34
         TYPE(MPI_Op), INTENT(IN) :: op
35
         TYPE(MPI_Win), INTENT(IN) :: win
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
39
                  target_disp, target_count, target_datatype, win, ierror)
40
                  BIND(C)
41
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
42
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
43
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
44
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
45
         TYPE(MPI_Win), INTENT(IN) :: win
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
48
```

1 target\_disp, target\_count, target\_datatype, op, win, request, 2 ierror) BIND(C) 3 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 4 INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 56 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 7 TYPE(MPI\_Op), INTENT(IN) :: op 8 TYPE(MPI\_Win), INTENT(IN) :: win TYPE(MPI\_Request), INTENT(OUT) :: 9 request 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI\_Rget(origin\_addr, origin\_count, origin\_datatype, target\_rank, 12target\_disp, target\_count, target\_datatype, win, request, 13 ierror) BIND(C) 14TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: origin\_addr 15INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 16 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 17INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 18 TYPE(MPI\_Win), INTENT(IN) :: win 19 TYPE(MPI\_Request), INTENT(OUT) :: request 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 23result\_addr, result\_count, result\_datatype, target\_rank, 24target\_disp, target\_count, target\_datatype, op, win, request, 25ierror) BIND(C) 26TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 27TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: result\_addr 28 INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 29target\_count TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype, 30 31result\_datatype 32 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 33 TYPE(MPI\_Op), INTENT(IN) :: op 34 TYPE(MPI\_Win), INTENT(IN) :: win 35 TYPE(MPI\_Request), INTENT(OUT) :: request 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI\_Rput(origin\_addr, origin\_count, origin\_datatype, target\_rank, 38 target\_disp, target\_count, target\_datatype, win, request, 39 ierror) BIND(C) 40 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 41 INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 42TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 43 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 44 TYPE(MPI\_Win), INTENT(IN) :: win 45TYPE(MPI\_Request), INTENT(OUT) :: request 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

```
1
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) BIND(C)
\mathbf{2}
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
4
         INTEGER, INTENT(IN) :: disp_unit
5
         TYPE(MPI_Info), INTENT(IN) :: info
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(C_PTR), INTENT(OUT) :: baseptr
8
         TYPE(MPI_Win), INTENT(OUT) :: win
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
11
                  BIND(C)
12
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
13
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
14
         INTEGER, INTENT(IN) :: disp_unit
15
         TYPE(MPI_Info), INTENT(IN) :: info
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         TYPE(C_PTR), INTENT(OUT) :: baseptr
18
         TYPE(MPI_Win), INTENT(OUT) :: win
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Win_attach(win, base, size, ierror) BIND(C)
22
         TYPE(MPI_Win), INTENT(IN) :: win
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
^{24}
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                         size
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Win_complete(win, ierror) BIND(C)
27
         TYPE(MPI_Win), INTENT(IN) :: win
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) BIND(C)
31
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
32
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
33
         INTEGER, INTENT(IN) :: disp_unit
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
     MPI_Win_create_dynamic(info, comm, win, ierror) BIND(C)
39
         TYPE(MPI_Info), INTENT(IN) ::
                                         info
40
         TYPE(MPI_Comm), INTENT(IN) ::
                                         comm
41
         TYPE(MPI_Win), INTENT(OUT) :: win
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Win_detach(win, base, ierror) BIND(C)
45
         TYPE(MPI_Win), INTENT(IN) :: win
46
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_Win_fence(assert, win, ierror) BIND(C)	1
INTEGER, INTENT(IN) :: assert TYPE(MPI_Win), INTENT(IN) :: win	2 3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	5
<pre>MPI_Win_flush(rank, win, ierror) BIND(C) INTEGER, INTENT(IN) :: rank</pre>	6
TYPE(MPI_Win), INTENT(IN) :: win	7 8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
MPI_Win_flush_all(win, ierror) BIND(C)	10
TYPE(MPI_Win), INTENT(IN) :: win	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12 13
MPI_Win_flush_local(rank, win, ierror) BIND(C)	13
INTEGER, INTENT(IN) :: rank	15
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
INTEGER, OPTIONAL, INTENT(UUT) :: Terror	17
MPI_Win_flush_local_all(win, ierror) BIND(C)	18 19
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21
<pre>MPI_Win_free(win, ierror) BIND(C)     TYPE(MPI_Win), INTENT(INOUT) :: win</pre>	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23 24
	25
<pre>MPI_Win_get_group(win, group, ierror) BIND(C)     TYPE(MPI_Win), INTENT(IN) :: win</pre>	26
TYPE(MPI_Group), INTENT(OUT) :: group	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28 29
MPI_Win_get_info(win, info_used, ierror) BIND(C)	29 30
TYPE(MPI_Win), INTENT(IN) :: win	31
TYPE(MPI_Info), INTENT(OUT) :: info_used	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
MPI_Win_lock(lock_type, rank, assert, win, ierror) BIND(C)	34 35
INTEGER, INTENT(IN) :: lock_type, rank, assert TYPE(MPI_Win), INTENT(IN) :: win	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
	38
<pre>MPI_Win_lock_all(assert, win, ierror) BIND(C) INTEGER, INTENT(IN) :: assert</pre>	39 40
TYPE(MPI_Win), INTENT(IN) :: win	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MPI_Win_post(group, assert, win, ierror) BIND(C)	43
TYPE(MPI_Group), INTENT(IN) :: group	44 45
INTEGER, INTENT(IN) :: assert	45 46
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
INIEGER, OFITOWAE, INTENT(OUT) TETTOT	48

```
1
     MPI_Win_set_info(win, info, ierror) BIND(C)
\mathbf{2}
         TYPE(MPI_Win), INTENT(IN) :: win
3
         TYPE(MPI_Info), INTENT(IN) :: info
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) BIND(C)
6
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7
         TYPE(MPI_Win), INTENT(IN) :: win
8
         INTEGER, INTENT(IN) :: rank
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
10
         INTEGER, INTENT(OUT) :: disp_unit
11
         TYPE(C_PTR), INTENT(OUT) :: baseptr
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Win_start(group, assert, win, ierror) BIND(C)
15
         TYPE(MPI_Group), INTENT(IN) :: group
16
         INTEGER, INTENT(IN) :: assert
17
         TYPE(MPI_Win), INTENT(IN) :: win
18
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
19
     MPI_Win_sync(win, ierror) BIND(C)
20
         TYPE(MPI_Win), INTENT(IN) :: win
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Win_test(win, flag, ierror) BIND(C)
^{24}
         TYPE(MPI_Win), INTENT(IN) :: win
25
         LOGICAL, INTENT(OUT) :: flag
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Win_unlock(rank, win, ierror) BIND(C)
28
         INTEGER, INTENT(IN) :: rank
29
         TYPE(MPI_Win), INTENT(IN) :: win
30
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
^{31}
32
     MPI_Win_unlock_all(win, ierror) BIND(C)
33
         TYPE(MPI_Win), INTENT(IN) :: win
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Win_wait(win, ierror) BIND(C)
36
         TYPE(MPI_Win), INTENT(IN) :: win
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
40
     A.3.10 External Interfaces Fortran 2008 Bindings
41
42
    MPI_Grequest_complete(request, ierror) BIND(C)
         TYPE(MPI_Request), INTENT(IN) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
    MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
46
                   ierror) BIND(C)
47
         PROCEDURE(MPI_Grequest_query_function) :: query_fn
48
```

PROCEDURE(MPI_Grequest_free_function) :: free_fn	1 2
PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	2
TYPE(MPI_Request), INTENT(OUT) :: request	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
	6
MPI_Init_thread(required, provided, ierror) BIND(C)	7
INTEGER, INTENT(IN) :: required	8
INTEGER, INTENT(OUT) :: provided	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Is_thread_main(flag, ierror) BIND(C)	11
LOGICAL, INTENT(OUT) :: flag	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Query_thread(provided, ierror) BIND(C)	14 15
INTEGER, INTENT(OUT) :: provided	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MDI Status set concelled(status flow isomer) BIND(C)	18
<pre>MPI_Status_set_cancelled(status, flag, ierror) BIND(C)         TYPE(MPI_Status), INTENT(INOUT) :: status</pre>	19
LOGICAL, INTENT(OUT) :: flag	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
	22
MPI_Status_set_elements(status, datatype, count, ierror) BIND(C)	23
TYPE(MPI_Status), INTENT(INOUT) :: status	24
TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: count	25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	26
	27
<pre>MPI_Status_set_elements_x(status, datatype, count, ierror) BIND(C)</pre>	28 29
TYPE(MPI_Status), INTENT(INOUT) :: status	30
TYPE(MPI_Datatype), INTENT(IN) :: datatype	31
INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) :: count	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
	34
A.3.11 I/O Fortran 2008 Bindings	35
MPI_File_close(fh, ierror) BIND(C)	36
TYPE(MPI_File), INTENT(INOUT) :: fh	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
	39
MPI_File_delete(filename, info, ierror) BIND(C)	40
CHARACTER(LEN=*), INTENT(IN) :: filename TYPE(MPI_Info), INTENT(IN) :: info	41 42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
	44
MPI_File_get_amode(fh, amode, ierror) BIND(C)	45
TYPE(MPI_File), INTENT(IN) :: fh	46
INTEGER, INTENT(OUT) :: amode	47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
    MPI_File_get_atomicity(fh, flag, ierror) BIND(C)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         LOGICAL, INTENT(OUT) :: flag
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
^{11}
    MPI_File_get_group(fh, group, ierror) BIND(C)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(MPI_Group), INTENT(OUT) :: group
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
    MPI_File_get_info(fh, info_used, ierror) BIND(C)
16
         TYPE(MPI_File), INTENT(IN) :: fh
17
         TYPE(MPI_Info), INTENT(OUT) :: info_used
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_File_get_position(fh, offset, ierror) BIND(C)
21
         TYPE(MPI_File), INTENT(IN) :: fh
22
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
    MPI_File_get_position_shared(fh, offset, ierror) BIND(C)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
^{29}
    MPI_File_get_size(fh, size, ierror) BIND(C)
30
         TYPE(MPI_File), INTENT(IN) :: fh
31
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_File_get_type_extent(fh, datatype, extent, ierror) BIND(C)
34
         TYPE(MPI_File), INTENT(IN) :: fh
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
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
43
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                  6
             BIND(C)
                                                                                  7
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  8
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 10
    INTEGER, INTENT(IN) :: count
                                                                                 11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 14
                                                                                 15
MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) BIND(C)
                                                                                 16
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 17
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 18
    INTEGER, INTENT(IN) :: count
                                                                                 19
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 20
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 22
MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)
                                                                                 23
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 24
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 25
    INTEGER, INTENT(IN) :: count
                                                                                 26
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 29
                                                                                 30
MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                 31
             BIND(C)
                                                                                 32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 33
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 34
    TYPE(*), DIMENSION(...), INTENT(IN), 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_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
                                                                                 40
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 41
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 42
    INTEGER, INTENT(IN) :: count
                                                                                 43
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 44
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 46
                                                                                 47
MPI_File_open(comm, filename, amode, info, fh, ierror) BIND(C)
                                                                                 48
```

751

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         CHARACTER(LEN=*), INTENT(IN) :: filename
3
         INTEGER, INTENT(IN) :: amode
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         TYPE(MPI_File), INTENT(OUT) :: fh
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
    MPI_File_preallocate(fh, size, ierror) BIND(C)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_File_read(fh, buf, count, datatype, status, ierror) BIND(C)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         TYPE(*), DIMENSION(..) :: 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
    MPI_File_read_all(fh, buf, count, datatype, status, ierror) BIND(C)
20
         TYPE(MPI_File), INTENT(IN) :: fh
21
         TYPE(*), DIMENSION(..) :: buf
22
         INTEGER, INTENT(IN) :: count
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Status) :: status
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
    MPI_File_read_all_begin(fh, buf, count, datatype, ierror) BIND(C)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
30
         INTEGER, INTENT(IN) :: count
^{31}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_File_read_all_end(fh, buf, status, ierror) BIND(C)
34
         TYPE(MPI_File), INTENT(IN) :: fh
35
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
36
         TYPE(MPI_Status) :: status
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
    MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
42
         TYPE(*), DIMENSION(..) :: buf
43
         INTEGER, INTENT(IN) :: count
44
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
         TYPE(MPI_Status) :: status
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
48
```

```
1
             BIND(C)
                                                                                 \mathbf{2}
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 3
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 4
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
                                                                                 5
                                                                                 6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 7
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 8
                                                                                 9
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
                                                                                 10
             BIND(C)
                                                                                 11
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 12
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 13
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 14
    INTEGER, INTENT(IN) :: count
                                                                                 15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 17
                                                                                 18
MPI_File_read_at_all_end(fh, buf, status, ierror) BIND(C)
                                                                                 19
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 20
                                                                                 21
    TYPE(MPI_Status) :: status
                                                                                 22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 23
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                 24
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 25
    TYPE(*), DIMENSION(..) :: buf
                                                                                 26
    INTEGER, INTENT(IN) :: count
                                                                                 27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 28
    TYPE(MPI_Status) :: status
                                                                                 29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 30
                                                                                 31
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
                                                                                 32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 33
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
                                                                                 34
    INTEGER, INTENT(IN) :: count
                                                                                 35
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 37
MPI_File_read_ordered_end(fh, buf, status, ierror) BIND(C)
                                                                                 38
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 40
    TYPE(MPI_Status) :: status
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
                                                                                 43
MPI_File_read_shared(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                 44
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 45
    TYPE(*), DIMENSION(..) :: buf
                                                                                 46
    INTEGER, INTENT(IN) :: count
                                                                                 47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 48
```

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_seek(fh, offset, whence, ierror) BIND(C)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
6
         INTEGER, INTENT(IN) :: whence
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_File_seek_shared(fh, offset, whence, ierror) BIND(C)
10
         TYPE(MPI_File), INTENT(IN) :: fh
11
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
12
         INTEGER, INTENT(IN) :: whence
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
    MPI_File_set_atomicity(fh, flag, ierror) BIND(C)
15
         TYPE(MPI_File), INTENT(IN) :: fh
16
         LOGICAL, INTENT(IN) :: flag
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
    MPI_File_set_info(fh, info, ierror) BIND(C)
20
         TYPE(MPI_File), INTENT(IN) :: fh
21
         TYPE(MPI_Info), INTENT(IN) :: info
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
    MPI_File_set_size(fh, size, ierror) BIND(C)
24
         TYPE(MPI_File), INTENT(IN) :: fh
25
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
    MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)
29
         TYPE(MPI_File), INTENT(IN) :: fh
30
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
^{31}
         TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
32
         CHARACTER(LEN=*), INTENT(IN) :: datarep
33
         TYPE(MPI_Info), INTENT(IN) :: info
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
    MPI_File_sync(fh, ierror) BIND(C)
36
         TYPE(MPI_File), INTENT(IN) :: fh
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
    MPI_File_write(fh, buf, count, datatype, status, ierror) BIND(C)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
42
         INTEGER, INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Status) :: status
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
    MPI_File_write_all(fh, buf, count, datatype, status, ierror) BIND(C)
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
                                                                                 1
                                                                                 2
    INTEGER, INTENT(IN) :: count
                                                                                 3
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
                                                                                 4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 5
                                                                                 6
MPI_File_write_all_begin(fh, buf, count, datatype, ierror) BIND(C)
                                                                                 7
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 8
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 9
    INTEGER, INTENT(IN) :: count
                                                                                 10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                12
                                                                                13
MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)
                                                                                14
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                16
    TYPE(MPI_Status) :: status
                                                                                17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                18
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
                                                                                19
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                20
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                21
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
                                                                                22
    INTEGER, INTENT(IN) :: count
                                                                                23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                24
    TYPE(MPI_Status) :: status
                                                                                25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                26
                                                                                27
MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
                                                                                28
             BIND(C)
                                                                                29
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                30
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                31
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
                                                                                32
    INTEGER, INTENT(IN) :: count
                                                                                33
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                34
    TYPE(MPI_Status) :: status
                                                                                35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                36
MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
                                                                                37
             BIND(C)
                                                                                38
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                39
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 40
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                41
    INTEGER, INTENT(IN) :: count
                                                                                42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                44
                                                                                45
MPI_File_write_at_all_end(fh, buf, status, ierror) BIND(C)
                                                                                46
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                47
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                48
```

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     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
11
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
14
         INTEGER, INTENT(IN) :: count
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_File_write_ordered_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_shared(fh, buf, count, datatype, status, ierror) BIND(C)
^{24}
         TYPE(MPI_File), INTENT(IN) :: fh
25
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
26
         INTEGER, INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Status) :: status
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
^{31}
                  dtype_file_extent_fn, extra_state, ierror) BIND(C)
32
         CHARACTER(LEN=*), INTENT(IN) :: datarep
33
         PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
34
         PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
35
         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
36
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
40
     A.3.12 Language Bindings Fortran 2008 Bindings
41
    MPI_F_sync_reg(buf) BIND(C)
42
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
43
44
    MPI_Sizeof(x, size, ierror) BIND(C)
45
         TYPE(*), DIMENSION(..) :: x
46
         INTEGER, INTENT(OUT) :: size
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

<pre>MPI_Status_f082f(f08_status, f_status, ierror) BIND(C)     TYPE(MPI_Status), INTENT(IN) :: f08_status     INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	1 2 3 4
<pre>MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C) INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE) TYPE(MPI_Status), INTENT(OUT) :: f08_status INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	5 6 7 8 9
<pre>MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C) INTEGER, INTENT(IN) :: p, r TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	10 11 12 13 14
<pre>MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C) INTEGER, INTENT(IN) :: r TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	14 15 16 17 18
<pre>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</pre>	19 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>	24 25 26 27
A.3.13 Tools / Profiling Interface Fortran 2008 Bindings MPI_Pcontrol(level) BIND(C)	28 29 30 31
INTEGER, INTENT(IN) :: level	32 33 34 35
	36 37 38
	39 40 41 42
	43 44 45 46
	46 47

A.4 Fortran Bindings with mpif.h or the mpi Module 1 $\mathbf{2}$ A.4.1 Point-to-Point Communication Fortran Bindings 3 4 MPI\_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 5<type> BUF(\*) 6 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 7 MPI\_BSEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 8 9 <type> BUF(\*) 10 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR  $^{11}$ MPI\_BUFFER\_ATTACH(BUFFER, SIZE, IERROR) 12<type> BUFFER(\*) 13INTEGER SIZE, IERROR 1415MPI\_BUFFER\_DETACH(BUFFER\_ADDR, SIZE, IERROR) 16<type> BUFFER\_ADDR(\*) 17INTEGER SIZE, IERROR 18MPI\_CANCEL(REQUEST, IERROR) 19 INTEGER REQUEST, IERROR 2021MPI\_GET\_COUNT(STATUS, DATATYPE, COUNT, IERROR) 22INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, COUNT, IERROR 23MPI\_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 24<type> BUF(\*) 25INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 2627MPI\_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) 28 INTEGER SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS(MPI\_STATUS\_SIZE), 29IERROR 30 MPI\_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)  $^{31}$ <type> BUF(\*) 32 INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR 33 34MPI\_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR) 35 LOGICAL FLAG 36 INTEGER SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR 37 MPI\_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(\*) 39 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 40 $^{41}$ MPI\_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 42<type> BUF(\*) 43INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 44MPI\_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 45<type> BUF(\*) 46INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4748

758

<pre><type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type></pre>	1 2 3
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)	4 5 6
<pre>MPI_MRECV(BOF, COUNT, DATATIPE, MESSAGE, STATUS, TERROR)</pre>	7 8 9
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)	10 11 12
<pre>MPI_RECV(BOF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, TERROR)</pre>	13 14 15 16 17
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) <pre> type&gt; BUF(*) typec count datatype cource tag comm deduest ierdor </pre>	18 19 20
INTEGER REQUEST, IERROR <sup>2</sup>	21 22 23
MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)       2         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR       2         LOCICAL FLAC       2	24 25 25 26
<pre>MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)</pre>	27 28 29 30
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	31 32 33
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR</type>	34 35 36 37
<pre>MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, A</type></pre>	38 39 40 41 42
<pre>MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR</type></pre>	43 44 45 46 47 48

```
1
    MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
\mathbf{2}
         <type> BUF(*)
3
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4
     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_START(REQUEST, IERROR)
12
         INTEGER REQUEST, IERROR
13
14
     MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
15
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
16
    MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
17
         LOGICAL FLAG
18
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
19
20
     MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
21
         LOGICAL FLAG
22
         INTEGER COUNT, ARRAY_OF_REQUESTS(*),
23
         ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
^{24}
     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
25
         LOGICAL FLAG
26
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
27
         IERROR
28
29
     MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
30
                   ARRAY_OF_STATUSES, IERROR)
31
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
32
         ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
33
    MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
34
         LOGICAL FLAG
35
         INTEGER STATUS(MPI_STATUS_SIZE), IERROR
36
37
     MPI_WAIT(REQUEST, STATUS, IERROR)
38
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
39
     MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
40
         INTEGER COUNT, ARRAY_OF_REQUESTS(*)
41
         INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
42
43
     MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
44
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
45
         IERROR
46
47
    MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
48
                   ARRAY_OF_STATUSES, IERROR)
```

INTEGER INCOUNT, ARRAY\_OF\_REQUESTS(\*), OUTCOUNT, ARRAY\_OF\_INDICES(\*), ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE,\*), IERROR

	5
A.4.2 Datatypes Fortran Bindings	4 5
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)	6
<pre><type> LOCATION(*)</type></pre>	7
INTEGER IERROR	8
INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS	9
	10
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	11
INTEGEN STRIUS(MII_STRIUS_SIZE), DRIVITE, COUNT, TEMUM	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(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)	17
<type> INBUF(*), OUTBUF(*)</type>	18
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR	19
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	20
POSITION, IERROR)	21
INTEGER INCOUNT, DATATYPE, IERROR	22
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION	23 24
CHARACTER*(*) DATAREP	24 25
<type> INBUF(*), OUTBUF(*)</type>	26
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)	27
INTEGER INCOUNT, DATATYPE, IERROR	28
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE	29
CHARACTER*(*) DATAREP	30
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)	31
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR	32
	33
MPI_TYPE_COMMIT(DATATYPE, IERROR)	34 35
INTEGER DATATYPE, IERROR	36
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)	37
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR	38
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,	39
ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,	40
OLDTYPE, NEWTYPE, IERROR)	41
INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),	42
ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR	43
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,	44
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	45
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR	46 47
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	47 48

1

 $\mathbf{2}$ 3

1 2 3 4	MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
5 6 7 8 9	MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
10 11 12 13 14	MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR
14 15 16 17	MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
18 19 20 21 22	<pre>MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>
23 24 25 26 27	MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
28 29 30	MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR
31 32	MPI_TYPE_FREE(DATATYPE, IERROR) INTEGER DATATYPE, IERROR
33 34 35 36 37 38	<pre>MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,</pre>
39 40 41 42 43	<pre>INTEGER(KIND=MP1_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*) MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,</pre>
44 45 46 47	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
48	MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)

INTEGER DATATYPE, IERROR INTEGER(KIND = MPI_COUNT_KIND) LB, EXTENT	1 2
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)	3 4
INTEGER DATATYPE, IERROR INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT	5 6
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR	7 8
INTEGER(KIND = MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT	9 10
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	11 12
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR	13 14
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR	15 16
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)	17 18
INTEGER DATATYPE, IERROR INTEGER(KIND = MPI_COUNT_KIND) SIZE	19 20
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	21 22 23
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR)	23 24 25
<type> INBUF(*), OUTBUF(*) INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR</type>	26 27
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, IERROR)	28 29
INTEGER OUTCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION	30 31
CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*)</type>	32 33
	34 35
A.4.3 Collective Communication Fortran Bindings	36
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)	37 38 39
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type>	40 41
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)	42 43
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,</type>	44 45
INTEGER SENDCOUNT, SENDTIFE, RECVCOUNTS(*), DISFLS(*), RECVTIFE, COMM, IERROR	46
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	47 48

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
3
     MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
4
                   COMM, IERROR)
5
         <type> SENDBUF(*), RECVBUF(*)
6
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
7
8
     MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
9
                   RDISPLS, RECVTYPE, COMM, IERROR)
10
         <type> SENDBUF(*), RECVBUF(*)
11
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
12
         RECVTYPE, COMM, IERROR
13
    MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
14
                   RDISPLS, RECVTYPES, COMM, IERROR)
15
         <type> SENDBUF(*), RECVBUF(*)
16
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
17
         RDISPLS(*), RECVTYPES(*), COMM, IERROR
18
19
     MPI_BARRIER(COMM, IERROR)
20
         INTEGER COMM, IERROR
21
    MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
22
         <type> BUFFER(*)
23
         INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
24
25
    MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
26
         <type> SENDBUF(*), RECVBUF(*)
27
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
28
     MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
29
                   ROOT, COMM, IERROR)
30
         <type> SENDBUF(*), RECVBUF(*)
^{31}
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
32
33
     MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
34
                   RECVTYPE, ROOT, COMM, IERROR)
35
         <type> SENDBUF(*), RECVBUF(*)
36
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
37
         COMM, IERROR
38
     MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
39
                   COMM, REQUEST, IERROR)
40
         <type> SENDBUF(*), RECVBUF(*)
41
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
42
43
     MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
44
                   RECVTYPE, COMM, REQUEST, IERROR)
45
         <type> SENDBUF(*), RECVBUF(*)
46
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
47
         REQUEST, IERROR
48
```

MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,	1
IERROR)	2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR	4
	5
MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	6
COMM, REQUEST, IERROR)	7
<type> SENDBUF(*), RECVBUF(*)</type>	8
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	9
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	10
RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)	11
<type> SENDBUF(*), RECVBUF(*)</type>	12
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	13
RECVTYPE, COMM, REQUEST, IERROR	14
	15
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	16
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)	17
<type> SENDBUF(*), RECVBUF(*)</type>	18
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),</pre>	19
RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR	20
	21
MPI_IBARRIER(COMM, REQUEST, IERROR)	22
INTEGER COMM, REQUEST, IERROR	22
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)	20
<pre><type> BUFFER(*)</type></pre>	24
INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR	25 26
MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)	27
<type> SENDBUF(*), RECVBUF(*)</type>	28
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR	29
	30
MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	31
ROOT, COMM, REQUEST, IERROR)	32
<pre><type> SENDBUF(*), RECVBUF(*) INTEGED GENDGOUNT DEGUTADE DOOT GOWN DEGUEGT</type></pre>	33
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	34
IERROR	35
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	36
RECVTYPE, ROOT, COMM, REQUEST, IERROR)	37
<type> SENDBUF(*), RECVBUF(*)</type>	38
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,	39
COMM, REQUEST, IERROR	40
	41
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,	42
IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR	45
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,	46
REQUEST, IERROR)	47
	48

765

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
3
     MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
4
                   REQUEST, IERROR)
5
         <type> SENDBUF(*), RECVBUF(*)
6
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
7
8
     MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
9
         <type> SENDBUF(*), RECVBUF(*)
10
         INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
11
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
12
                   ROOT, COMM, REQUEST, IERROR)
13
         <type> SENDBUF(*), RECVBUF(*)
14
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
15
         IERROR
16
17
     MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
18
                   RECVTYPE, ROOT, COMM, REQUEST, IERROR)
19
         <type> SENDBUF(*), RECVBUF(*)
20
         INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
21
         COMM, REQUEST, IERROR
22
     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
23
         LOGICAL COMMUTE
24
         INTEGER OP, IERROR
25
26
     MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
27
         EXTERNAL USER_FN
28
         LOGICAL COMMUTE
29
         INTEGER OP, IERROR
30
    MPI_OP_FREE(OP, IERROR)
^{31}
         INTEGER OP, IERROR
32
33
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
34
         <type> SENDBUF(*), RECVBUF(*)
35
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
36
     MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
37
         <type> INBUF(*), INOUTBUF(*)
38
         INTEGER COUNT, DATATYPE, OP, IERROR
39
40
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
41
                   IERROR)
42
         <type> SENDBUF(*), RECVBUF(*)
43
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
44
    MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
45
                   IERROR)
46
         <type> SENDBUF(*), RECVBUF(*)
47
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
48
```

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	1 2
<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR</type>	3
<pre>MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR</type></pre>	4 5 6 7 8
<pre>MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,</pre>	9 10 11 12 13 14 15
A.4.4 Groups, Contexts, Communicators, and Caching Fortran Bindings	16
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) INTEGER COMM1, COMM2, RESULT, IERROR	17 18 19
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR) INTEGER COMM, GROUP, NEWCOMM, IERROR	20 21
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR	22 23 24
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL, EXTRA_STATE, IERROR) EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN INTEGER COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	24 25 26 27 28 29
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR	30 31 32
MPI_COMM_DUP(COMM, NEWCOMM, IERROR) INTEGER COMM, NEWCOMM, IERROR	33 34
<pre>MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>	35 36 37 38 39 40 41
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR) INTEGER COMM, INFO, NEWCOMM, IERROR	42 43
MPI_COMM_FREE(COMM, IERROR) INTEGER COMM, IERROR	$44 \\ 45 \\ 46$
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR) INTEGER COMM_KEYVAL, IERROR	47 48

```
1
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
\mathbf{2}
         INTEGER COMM, COMM_KEYVAL, IERROR
3
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
4
         LOGICAL FLAG
5
     MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
6
         INTEGER COMM, INFO USED, IERROR
\overline{7}
8
     MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
9
         INTEGER COMM, RESULTLEN, IERROR
10
         CHARACTER*(*) COMM_NAME
11
     MPI_COMM_GROUP(COMM, GROUP, IERROR)
12
         INTEGER COMM, GROUP, IERROR
13
14
     MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
15
         INTEGER COMM, NEWCOMM, REQUEST, IERROR
16
     MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
17
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
18
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
19
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
20
             ATTRIBUTE_VAL_OUT
21
         LOGICAL FLAG
22
23
     MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
^{24}
                   IERROR)
25
         INTEGER COMM, COMM_KEYVAL, IERROR
26
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
27
     MPI_COMM_RANK(COMM, RANK, IERROR)
28
         INTEGER COMM, RANK, IERROR
29
30
     MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
^{31}
         INTEGER COMM, GROUP, IERROR
32
     MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
33
         INTEGER COMM, SIZE, IERROR
34
35
     MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
36
         INTEGER COMM, COMM_KEYVAL, IERROR
37
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
38
39
     MPI_COMM_SET_INFO(COMM, INFO, IERROR)
         INTEGER COMM, INFO, IERROR
40
41
     MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
42
         INTEGER COMM, IERROR
43
         CHARACTER*(*) COMM_NAME
44
     MPI_COMM_SIZE(COMM, SIZE, IERROR)
45
46
         INTEGER COMM, SIZE, IERROR
47
     MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
48
```

INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	1 2
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	3
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR	4
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)	5
INTEGER COMM, IERROR	6
LOGICAL FLAG	7
	8
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)	9
INTEGER GROUP1, GROUP2, RESULT, IERROR	10
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)	11
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	12
	13
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	14
INTEGER GROOF, N, RANKS(*), NEWGROOF, IERROR	15
MPI_GROUP_FREE(GROUP, IERROR)	16
INTEGER GROUP, IERROR	17 18
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)	19
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	20
	21
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)	22
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	23
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)	24
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR	25
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)	26
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR	27
	28
MPI_GROUP_RANK(GROUP, RANK, IERROR)	29
INTEGER GROUP, RANK, IERROR	30
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)	31
INTEGER GROUP, SIZE, IERROR	32 33
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)	33 34
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR	35
	36
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)	37
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	38
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,	39
TAG, NEWINTERCOMM, IERROR)	40
INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,	41
NEWINTERCOMM, IERROR	42
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)	43
INTEGER INTERCOMM, NEWINTRACOMM, IERROR	44
LOGICAL HIGH	45
	46 47
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,	47 48
	40

1	EXTRA_STATE, IERROR)
2	EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
3	INTEGER TYPE_KEYVAL, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
5 6	MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
7	INTEGER DATATYPE, TYPE_KEYVAL, IERROR
8	MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
9	ATTRIBUTE_VAL_OUT, FLAG, IERROR)
10	INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
11	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
12	ATTRIBUTE_VAL_OUT
13	LOGICAL FLAG
14 15	MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
16	INTEGER TYPE_KEYVAL, IERROR
17	MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
18	INTEGER DATATYPE, TYPE_KEYVAL, IERROR
19	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
20	LOGICAL FLAG
21	
22	MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR
23 24	CHARACTER*(*) TYPE_NAME
24	
25	
25 26	MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
	ATTRIBUTE_VAL_OUT, FLAG, IERROR)
26	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
26 27	ATTRIBUTE_VAL_OUT, FLAG, IERROR)
26 27 28 29 30	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
26 27 28 29 30 31	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG
26 27 28 29 30 31 32	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
26 27 28 29 30 31	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG
26 27 28 29 30 31 32 33	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
26 27 28 29 30 31 32 33 34	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
26 27 28 29 30 31 32 33 34 35	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
26 27 28 29 30 31 32 33 34 35 36 37 38	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
26 27 28 29 30 31 32 33 34 35 36 37 38 39	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
26 27 28 30 31 32 33 34 35 36 37 38 39 40	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
26 27 28 29 30 31 32 33 34 35 36 37 38 39	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, IERROR) INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL, EXTRA_STATE, IERROR)
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL, EXTRA_STATE, IERROR) EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, IERROR) INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL, EXTRA_STATE, IERROR)

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	771
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR	1 2
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDWIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG	3 4 5 6 7 8 9
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR) INTEGER WIN_KEYVAL, IERROR	10 11
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL LOGICAL FLAG	12 13 14 15 16
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR) INTEGER WIN, RESULTLEN, IERROR CHARACTER*(*) WIN_NAME	17 18 19 20
<pre>MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>	20 21 22 23 24 25 26
MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR INTEGER WIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	28 29
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	30 31 32 33
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR) INTEGER WIN, IERROR CHARACTER*(*) WIN_NAME	34 35 36 37 38
A.4.5 Process Topologies Fortran Bindings	39
MPI_CARTDIM_GET(COMM, NDIMS, IERROR) INTEGER COMM, NDIMS, IERROR	40 41 42
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR) INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR	43 44
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERR INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR LOGICAL PERIODS(*), REORDER	45 OR) 46 47 48

1 MPI\_CART\_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)  $\mathbf{2}$ INTEGER COMM, MAXDIMS, DIMS(\*), COORDS(\*), IERROR 3 LOGICAL PERIODS(\*) 4 MPI\_CART\_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) 5INTEGER COMM, NDIMS, DIMS(\*), NEWRANK, IERROR 6 LOGICAL PERIODS(\*) 7 8 MPI\_CART\_RANK(COMM, COORDS, RANK, IERROR) 9 INTEGER COMM, COORDS(\*), RANK, IERROR 10 MPI\_CART\_SHIFT(COMM, DIRECTION, DISP, RANK\_SOURCE, RANK\_DEST, IERROR) 11 INTEGER COMM, DIRECTION, DISP, RANK\_SOURCE, RANK\_DEST, IERROR 1213MPI\_CART\_SUB(COMM, REMAIN\_DIMS, NEWCOMM, IERROR) 14INTEGER COMM, NEWCOMM, IERROR 15LOGICAL REMAIN\_DIMS(\*) 16MPI\_DIMS\_CREATE(NNODES, NDIMS, DIMS, IERROR) 17INTEGER NNODES, NDIMS, DIMS(\*), IERROR 18 19MPI\_DIST\_GRAPH\_CREATE(COMM\_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS, 20INFO, REORDER, COMM\_DIST\_GRAPH, IERROR) 21INTEGER COMM\_OLD, N, SOURCES(\*), DEGREES(\*), DESTINATIONS(\*), 22WEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR 23LOGICAL REORDER  $^{24}$ MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(COMM\_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, 25OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, 26COMM\_DIST\_GRAPH, IERROR) 27INTEGER COMM\_OLD, INDEGREE, SOURCES(\*), SOURCEWEIGHTS(\*), OUTDEGREE, 28DESTINATIONS(\*), DESTWEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR 29LOGICAL REORDER 30 31MPI\_DIST\_GRAPH\_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, 32 MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) 33 INTEGER COMM, MAXINDEGREE, SOURCES(\*), SOURCEWEIGHTS(\*), MAXOUTDEGREE, 34 DESTINATIONS(\*), DESTWEIGHTS(\*), IERROR 35 MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) 36 INTEGER COMM, INDEGREE, OUTDEGREE, IERROR 37 LOGICAL WEIGHTED 3839 MPI\_GRAPHDIMS\_GET(COMM, NNODES, NEDGES, IERROR) 40INTEGER COMM, NNODES, NEDGES, IERROR 41 MPI\_GRAPH\_CREATE(COMM\_OLD, NNODES, INDEX, EDGES, REORDER, COMM\_GRAPH, 42IERROR) 43 INTEGER COMM\_OLD, NNODES, INDEX(\*), EDGES(\*), COMM\_GRAPH, IERROR 44LOGICAL REORDER 4546MPI\_GRAPH\_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) 47INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(\*), EDGES(\*), IERROR 48

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 773 MPI\_GRAPH\_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 1 INTEGER COMM, NNODES, INDEX(\*), EDGES(\*), NEWRANK, IERROR 2 MPI\_GRAPH\_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) 4 INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(\*), IERROR 56 MPI\_GRAPH\_NEIGHBORS\_COUNT(COMM, RANK, NNEIGHBORS, IERROR) 7 INTEGER COMM, RANK, NNEIGHBORS, IERROR 8 MPI\_INEIGHBOR\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 9 RECVTYPE, COMM, REQUEST, IERROR) 10<tvpe> SENDBUF(\*), RECVBUF(\*) 11 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 1213 MPI\_INEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 14DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 15<type> SENDBUF(\*), RECVBUF(\*) 16INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM. 17REQUEST, IERROR 18 MPI\_INEIGHBOR\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 19 RECVTYPE, COMM, REQUEST, IERROR) 20<type> SENDBUF(\*), RECVBUF(\*) 21INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 22 23MPI\_INEIGHBOR\_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 24RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 25<type> SENDBUF(\*), RECVBUF(\*) 26INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*). 27RECVTYPE, COMM, REQUEST, IERROR 28MPI\_INEIGHBOR\_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 29 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 30 <type> SENDBUF(\*), RECVBUF(\*) 31INTEGER(KIND=MPI\_ADDRESS\_KIND) SDISPLS(\*), RDISPLS(\*) 32 INTEGER SENDCOUNTS(\*), SENDTYPES(\*), RECVCOUNTS(\*), RECVTYPES(\*), COMM, 33 REQUEST, IERROR 34 35MPI\_NEIGHBOR\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 36 RECVTYPE, COMM, IERROR) 37 <type> SENDBUF(\*), RECVBUF(\*) 38INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 39 MPI\_NEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 40DISPLS, RECVTYPE, COMM, IERROR) 41 <type> SENDBUF(\*), RECVBUF(\*) 42INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 43 IERROR 4445MPI\_NEIGHBOR\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 46RECVTYPE, COMM, IERROR) 47<type> SENDBUF(\*), RECVBUF(\*) 48

```
1
                   INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
          \mathbf{2}
               MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
          3
                             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
          4
                   <type> SENDBUF(*), RECVBUF(*)
          5
                   INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
          6
                   RECVTYPE, COMM, IERROR
          7
          8
               MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
          9
                             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
          10
                   <type> SENDBUF(*), RECVBUF(*)
          11
                   INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
          12
                   INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
          13
                   IERROR
          14
               MPI_TOPO_TEST(COMM, STATUS, IERROR)
          15
                   INTEGER COMM, STATUS, IERROR
          16
          17
          18
               A.4.6 MPI Environmental Management Fortran Bindings
          19
               DOUBLE PRECISION MPI_WTICK()
          20
          21
               DOUBLE PRECISION MPI_WTIME()
          22
               MPI_ABORT(COMM, ERRORCODE, IERROR)
          23
                   INTEGER COMM, ERRORCODE, IERROR
          ^{24}
          25
               MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
          26
                   INTEGER ERRORCLASS, IERROR
          27
          28
               MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
                   INTEGER ERRORCLASS, ERRORCODE, IERROR
          29
          30
               MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
          ^{31}
                   INTEGER ERRORCODE, IERROR
          32
                   CHARACTER*(*) STRING
          33
          34
               MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
          35
                   INTEGER INFO. IERROR
          36
                   INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
ticket229.9.
          37
               If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
          38
                 INTERFACE MPI_ALLOC_MEM
          39
                   SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
          40
                     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
          41
                     INTEGER :: INFO, IERROR
          42
                     INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
          43
                     TYPE(C_PTR) :: BASEPTR
          44
                   END SUBROUTINE
          45
                 END INTERFACE
          46
          47
               MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
          48
                   INTEGER COMM, ERRORCODE, IERROR
```

MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL COMM_ERRHANDLER_FN	1 2 3
INTEGER ERRHANDLER, IERROR	3 4
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	5 6
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	7 8
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR) INTEGER ERRHANDLER, IERROR	9 10 11
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR) INTEGER ERRORCODE, ERRORCLASS, IERROR	11 12 13
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)	14 15
INTEGER ERRORCODE, RESULTLEN, IERROR CHARACTER*(*) STRING	16 17
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR) INTEGER FH, ERRORCODE, IERROR	18 19 20
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL FILE_ERRHANDLER_FN	20 21 22
INTEGER ERRHANDLER, IERROR MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	23 24
INTEGER FILE, ERRHANDLER, IERROR	25 26
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR) INTEGER FILE, ERRHANDLER, IERROR	27 28
MPI_FINALIZE(IERROR) INTEGER IERROR	29 30 31
MPI_FINALIZED(FLAG, IERROR) LOGICAL FLAG	32 33
INTEGER IERROR MPI_FREE_MEM(BASE, IERROR)	$34 \\ 35$
<pre><type> BASE(*) INTEGER IERROR</type></pre>	36 37
MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR) CHARACTER*(*) VERSION	38 39 40
INTEGER RESULTLEN, IERROR MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)	41 42
CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR	43 44
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)	45 46 47
INTEGER VERSION, SUBVERSION, IERROR	48

```
1
     MPI_INIT(IERROR)
\mathbf{2}
         INTEGER IERROR
3
     MPI_INITIALIZED(FLAG, IERROR)
4
         LOGICAL FLAG
\mathbf{5}
         INTEGER IERROR
6
7
     MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
8
         INTEGER WIN, ERRORCODE, IERROR
9
     MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
10
         EXTERNAL WIN_ERRHANDLER_FN
11
         INTEGER ERRHANDLER, IERROR
12
13
     MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
14
         INTEGER WIN, ERRHANDLER, IERROR
15
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
16
         INTEGER WIN, ERRHANDLER, IERROR
17
18
19
     A.4.7 The Info Object Fortran Bindings
20
21
     MPI_INFO_CREATE(INFO, IERROR)
         INTEGER INFO, IERROR
22
23
     MPI_INFO_DELETE(INFO, KEY, IERROR)
^{24}
         INTEGER INFO, IERROR
25
         CHARACTER*(*) KEY
26
     MPI_INFO_DUP(INFO, NEWINFO, IERROR)
27
28
         INTEGER INFO, NEWINFO, IERROR
^{29}
     MPI_INFO_FREE(INFO, IERROR)
30
         INTEGER INFO, IERROR
^{31}
32
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
33
         INTEGER INFO, VALUELEN, IERROR
34
         CHARACTER*(*) KEY, VALUE
35
         LOGICAL FLAG
36
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
37
         INTEGER INFO, NKEYS, IERROR
38
39
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
40
         INTEGER INFO, N, IERROR
41
         CHARACTER*(*) KEY
42
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
43
         INTEGER INFO, VALUELEN, IERROR
44
         LOGICAL FLAG
45
         CHARACTER*(*) KEY
46
47
     MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
48
```

INTEGER INFO, IERROR	1
CHARACTER*(*) KEY, VALUE	3
	4
A.4.8 Process Creation and Management Fortran Bindings	5
MPI_CLOSE_PORT(PORT_NAME, IERROR)	67
CHARACTER*(*) PORT_NAME INTEGER IERROR	8
	9
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)	10
CHARACTER*(*) PORT_NAME INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR	11
	13
<pre>MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR) CHARACTER*(*) PORT_NAME</pre>	14
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR	15
MPI_COMM_DISCONNECT(COMM, IERROR)	16 17
INTEGER COMM, IERROR	17
MPI_COMM_GET_PARENT(PARENT, IERROR)	19
INTEGER PARENT, IERROR	20
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)	21
INTEGER FD, INTERCOMM, IERROR	22
	24
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM ARRAY_OF_ERRCODES, IERROR)	P1, 25
CHARACTER*(*) COMMAND, ARGV(*)	26
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCOD	ES(*), 27
IERROR	29
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,	30
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCO	-
ARRAY_OF_ERRCODES, IERROR) INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, C	32 ∩MM
INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR	34
CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)	35
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	36
CHARACTER*(*) SERVICE_NAME, PORT_NAME	37 38
INTEGER INFO, IERROR	39
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)	40
CHARACTER*(*) PORT_NAME	41
INTEGER INFO, IERROR	42
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	44
INTEGER INFO, IERROR	45
CHARACTER*(*) SERVICE_NAME, PORT_NAME	46
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	47 48
INTEGER INFO, IERROR	40

```
1
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
2
3
     A.4.9 One-Sided Communications Fortran Bindings
4
\mathbf{5}
     MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
6
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, 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, OP, WIN, IERROR
11
    MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
12
                   TARGET_RANK, TARGET_DISP, WIN, IERROR)
13
         <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
14
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
15
         INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
16
17
     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
18
                   TARGET_DISP, OP, WIN, IERROR)
19
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
20
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
21
         INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
22
    MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
23
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
24
         <type> ORIGIN_ADDR(*)
25
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
26
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
27
         TARGET_DATATYPE, WIN, IERROR
28
29
     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
30
                   RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
31
                   TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
32
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
33
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
34
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
35
         TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
36
    MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
37
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
38
         <type> ORIGIN_ADDR(*)
39
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
40
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
41
         TARGET_DATATYPE, WIN, IERROR
42
43
     MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
44
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
45
                   IERROR)
46
         <type> ORIGIN_ADDR(*)
47
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
48
```

INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	1
IARGEI_DAIAIIFE, OF, WIN, REQUESI, IERROR	3
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	4
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,	5
IERROR)	6
<type> ORIGIN_ADDR(*)</type>	7
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	8
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	9
TARGET_DATATYPE, WIN, REQUEST, IERROR	
	10
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,	11
RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,	12
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	13
IERROR)	14
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	15
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	16
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,	17
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	18
	19
MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	20
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,	21
IERROR)	22
<type> ORIGIN_ADDR(*)</type>	23
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	24
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	25
TARGET_DATATYPE, WIN, REQUEST, IERROR	26
MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	27
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR	28
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	29
	$_{30}^{29}$ ticket229.9.
If the Fortran compiler provides TYPE(C_PTR), then overloaded by:	31
INTERFACE MPI_WIN_ALLOCATE	32
SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &	33
WIN, IERROR)	34
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	35
INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR	36
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	37
TYPE(C_PTR) :: BASEPTR	38
END SUBROUTINE	39
END INTERFACE	40
	41
MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	41
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR	42
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	$^{43}_{44}$ ticket229.9.
If the Fortran compiler provides TYPE(C_PTR), then overloaded by:	**
INTERFACE MPI_WIN_ALLOCATE_SHARED	45
SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &	46
BASEPTR, WIN, IERROR)	47
	48

```
1
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
\mathbf{2}
           INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
3
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
4
           TYPE(C_PTR) :: BASEPTR
5
         END SUBROUTINE
6
       END INTERFACE
7
     MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
8
         INTEGER WIN, IERROR
9
         <type> BASE(*)
10
         INTEGER (KIND=MPI_ADDRESS_KIND) SIZE
11
12
     MPI_WIN_COMPLETE(WIN, IERROR)
13
         INTEGER WIN, IERROR
14
     MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
15
         <type> BASE(*)
16
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
17
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
18
19
     MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
20
         INTEGER INFO, COMM, WIN, IERROR
21
     MPI_WIN_DETACH(WIN, BASE, IERROR)
22
         INTEGER WIN, IERROR
23
         <type> BASE(*)
^{24}
25
     MPI_WIN_FENCE(ASSERT, WIN, IERROR)
26
         INTEGER ASSERT, WIN, IERROR
27
     MPI_WIN_FLUSH(RANK, WIN, IERROR)
28
         INTEGER RANK, WIN, IERROR
29
30
     MPI_WIN_FLUSH_ALL(WIN, IERROR)
^{31}
         INTEGER WIN, IERROR
32
     MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
33
34
         INTEGER RANK, WIN, IERROR
35
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
36
         INTEGER WIN, IERROR
37
     MPI_WIN_FREE(WIN, IERROR)
38
39
         INTEGER WIN, IERROR
40
     MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
41
         INTEGER WIN, GROUP, IERROR
42
43
     MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
44
         INTEGER WIN, INFO_USED, IERROR
45
     MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
46
         INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
47
48
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
```

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	781
INTEGER ASSERT, WIN, IERROR	1
	2
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)	3
INTEGER GROUP, ASSERT, WIN, IERROR	4
MPI_WIN_SET_INFO(WIN, INFO, IERROR)	5
INTEGER WIN, INFO, IERROR	6
	7
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP_UNIT, IERROR	8
INTEGER WIN, RANK, DISP_UNII, IERRUR INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	9
-	$^{10}$ ticket 229.9.
If the Fortran compiler provides TYPE(C_PTR), then overloaded by:	11
INTERFACE MPI_WIN_SHARED_QUERY	12
SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &	13
BASEPTR, IERROR)	14
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	15
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	16 17
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	18
TYPE(C_PTR) :: BASEPTR END SUBROUTINE	19
END SUBRUUTINE END INTERFACE	20
END INTERFACE	20
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)	22
INTEGER GROUP, ASSERT, WIN, IERROR	23
MPI_WIN_SYNC(WIN, IERROR)	24
INTEGER WIN, IERROR	25
INTEGER WIN, IERROR	26
MPI_WIN_TEST(WIN, FLAG, IERROR)	27
INTEGER WIN, IERROR	28
LOGICAL FLAG	29
MPI_WIN_UNLOCK(RANK, WIN, IERROR)	30
INTEGER RANK, WIN, IERROR	31
	32
MPI_WIN_UNLOCK_ALL(WIN, IERROR)	33
INTEGER WIN, IERROR	34
MPI_WIN_WAIT(WIN, IERROR)	35
INTEGER WIN, IERROR	36
	37
A 4.10 Estevent later from Division	38 39
A.4.10 External Interfaces Fortran Bindings	40
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)	40
INTEGER REQUEST, IERROR	42
MDT ODENHEOT CTADT (NHEDV EN EDEE EN CAMCEL EN EVTDA CTATE DECHEOT	43
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR)	44
INTEGER REQUEST, IERROR	45
EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN	46
INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE	47
	48

$\frac{1}{2}$	MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR) INTEGER REQUIRED, PROVIDED, IERROR
3 4 5 6	MPI_IS_THREAD_MAIN(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR
7 8	MPI_QUERY_THREAD(PROVIDED, IERROR) INTEGER PROVIDED, IERROR
9 10 11 12	MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG
13 14	MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
15 16 17 18 19	MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER (KIND=MPI_COUNT_KIND) COUNT
20 21	A.4.11 I/O Fortran Bindings
22 23	MPI_FILE_CLOSE(FH, IERROR) INTEGER FH, IERROR
24 25 26 27	MPI_FILE_DELETE(FILENAME, INFO, IERROR) CHARACTER*(*) FILENAME INTEGER INFO, IERROR
28 29	MPI_FILE_GET_AMODE(FH, AMODE, IERROR) INTEGER FH, AMODE, IERROR
30 31 32 33	MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG
34 35 36	MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
37 38 39	MPI_FILE_GET_GROUP(FH, GROUP, IERROR) INTEGER FH, GROUP, IERROR
40 41	MPI_FILE_GET_INFO(FH, INFO_USED, IERROR) INTEGER FH, INFO_USED, IERROR
42 43 44 45	MPI_FILE_GET_POSITION(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
46 47 48	MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR) INTEGER FH, IERROR

INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	1
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	2 3
INTEGER FH, IERROR	4
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	5
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)	6
INTEGER FH, DATATYPE, IERROR	7
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT	8
	9
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)	10
INTEGER FH, ETYPE, FILETYPE, IERROR CHARACTER*(*) DATAREP	11
INTEGER(KIND=MPI_OFFSET_KIND) DISP	12
	13 14
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	15
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR</type>	16
INIEGER FII, COUNT, DATATITE, REQUEST, TERROR	17
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	18
<type> BUF(*)</type>	19
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	20
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	21
MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	22
<type> BUF(*)</type>	23 24
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	24
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	26
<type> BUF(*)</type>	27
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	28
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	29
<pre><type> BUF(*)</type></pre>	30
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	31
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	32
MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	33 34
<pre><type> BUF(*)</type></pre>	35
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	36
	37
<pre>MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) CHARACTER*(*) FILENAME</pre>	38
INTEGER COMM, AMODE, INFO, FH, IERROR	39
	40
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)	41
INTEGER FH, IERROR	42
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	43
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	44 45
<type> BUF(*)</type>	46
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	47
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	48

$\frac{1}{2}$	<pre><type> BUF(*) INTEGED EN COUNT DATATIVE CTATUS(NEL CTATUS (LEE) LEEDOD</type></pre>
3	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
4	MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
5	<type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type>
6 7	
8	<pre>MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)</pre>
9	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
10 11	MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
12	<type> BUF(*)</type>
$\frac{13}{14}$	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
15	MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
16 17	<pre><type> BUF(*) INTEGED EN COUNT DATATIVE CTATUS(NEL CTATUS CLEED) IEDDOD</type></pre>
18 19	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
20	MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
21	<type> BUF(*)</type>
22 23	INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
24	MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
25	<pre><type> BUF(*)</type></pre>
26 27	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
28	MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
29	<type> BUF(*)</type>
30	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
31 32	MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
33	<type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type>
34	
35 36	<pre>MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)</pre>
37	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
38	MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
39	<pre><type> BUF(*)</type></pre>
40 41	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
42	MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
43	INTEGER FH, WHENCE, IERROR
44	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45 46	MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
47	INTEGER FH, WHENCE, IERROR
48	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG	1 2 3
MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR	4 5 6
MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	7 8 9
<pre>MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR CHARACTER*(*) DATAREP INTEGER(KIND=MPI_OFFSET_KIND) DISP</pre>	10 11 12 13 14
MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR	15 16 17
<pre>MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	18 19 20
<pre>MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	21 22 23
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type>	24 25 26 27
<pre>MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)</pre>	28 29 30
<pre>MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)         <type> BUF(*)         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET</type></pre>	31 32 33 34 35
<pre>MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)         <type> BUF(*)         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET</type></pre>	36 37 38 39 40
<pre>MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)</pre>	40 41 42 43 44
<pre>MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)      <type> BUF(*)      INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type></pre>	45 46 47 48

```
1
     MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
\mathbf{2}
         <type> BUF(*)
3
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
4
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
5
         <type> BUF(*)
6
         INTEGER FH, COUNT, DATATYPE, IERROR
7
8
     MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
9
         <type> BUF(*)
10
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
11
    MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
12
         <type> BUF(*)
13
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
14
15
    MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
16
                   DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
17
         CHARACTER*(*) DATAREP
18
         EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
19
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
20
         INTEGER IERROR
21
22
     A.4.12 Language Bindings Fortran Bindings
23
^{24}
    MPI_F_SYNC_REG(buf)
25
         <type> buf(*)
26
     MPI_SIZEOF(X, SIZE, IERROR)
27
         <type> X
28
         INTEGER SIZE, IERROR
29
30
     MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
^{31}
         TYPE(MPI_Status) :: F08_STATUS
32
         INTEGER :: F_STATUS(MPI_STATUS_SIZE)
33
         INTEGER IERROR
34
    MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
35
         INTEGER :: F_STATUS(MPI_STATUS_SIZE)
36
37
         TYPE(MPI_Status) :: F08_STATUS
38
         INTEGER IERROR
39
    MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
40
         INTEGER P, R, NEWTYPE, IERROR
41
42
    MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
43
         INTEGER R, NEWTYPE, IERROR
44
     MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
45
         INTEGER P, R, NEWTYPE, IERROR
46
47
    MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
48
         INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

A.4.13 Tools / Profiling Interface Fortran Bindings	1
MPI_PCONTROL(LEVEL)	2
INTEGER LEVEL	3 4
	4 5
A.4.14 Deprecated Fortran Bindings	6
A.4.14 Deprecated Fortran Dindings	7
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)	8
INTEGER COMM, KEYVAL, IERROR	9
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	10
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	11
LOGICAL FLAG	12 13
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)	13
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	15
	16
MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)	17
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	18
ATTRIBUTE_VAL_OUT, IERR	19
LOGICAL FLAG	20
	21
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR) EXTERNAL COPY_FN, DELETE_FN	22 23
INTEGER KEYVAL, EXTRA_STATE, IERROR	23 24
	24
MPI_KEYVAL_FREE(KEYVAL, IERROR)	26
INTEGER KEYVAL, IERROR	27
MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	28
ATTRIBUTE_VAL_OUT, FLAG, IERR)	29
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	30
ATTRIBUTE_VAL_OUT, IERR	31
LOGICAL FLAG	32
MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)	33 34
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR	35
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	36
ATTRIBUTE_VAL_OUT, FLAG, IERR)	37
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	38
ATTRIBUTE_VAL_OUT, IERR	39
LOGICAL FLAG	40
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)	41
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	42
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,, _,, _	43
	44 45
	45
	-0

ticket281. $\frac{1}{2}$	
	[ includeappLang-C++ ]
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13 14	
14	
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	
45 46	
40	
48	
-0	

## 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
0.1		21
B.1.	1 Fixes to Errata in Previous Versions of MPI	22 23
1.	Sections $2.6.2$ and $2.6.3$ on pages 19 and 19, and	24
	MPI-2.2 Section 2.6.2 on page 17, lines 41-42, Section 2.6.3 on page 18, lines 15-16, and Section 2.6.4 on page 18, lines 40-41.	25 26
	This is an MPI-2 erratum: The scope for the reserved prefix MPI_ and the C++ namespace MPI is now any name as originally intended in MPI-1.	27 28
	hamespace in 1 is now any name as originary intended in wir F1.	29
2.	Sections 3.2.2, 5.9.2, 13.7.2 Table 13.2, and Annex A.1.1 on pages 25, 176, 536, and	30
	665, and	31
	MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1	32
	on pages 27, 164, 433, 472 and 513	33
	This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL,	34
	MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and	35
	MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding	36
	to the C++ types bool, std::complex <float>, std::complex<double>, and</double></float>	37
	std::complex <long double="">. These datatypes also correspond to the deprecated</long>	38
	C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX,	39
	and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-	40
	standard C++ types Complex<> were substituted by the standard types	41
	<pre>std::complex&lt;&gt;.</pre>	42
9	Sections 5.9.2 on pages 176 and MPI-2.2 Section 5.9.2, page 165, line 47.	43
э.	This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduc-	44
	tion group.	45
		46
4.	Section 7.5.5 on page 302, and	47
	MPI-2.2, Section 7.5.5 on page $257$ , C++ interface on page $264$ , line 3.	48

1 2 3	This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as int& indegree and int& outdegree in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.
4 5 6 7 8	<ol> <li>Section 13.7.2, Table 13.2 on page 536, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433. This was an MPI-2.2 erratum: The MPI_C_BOOL "external32" representation is corrected to a 1-byte size.</li> </ol>
9 10 11 12	<ol> <li>MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.</li> </ol>
13 14 15 16 17 18	<ol> <li>Annex A.1.1 on page 665, 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 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.</li> </ol>
19	B.1.2 Changes in MPI-3.0
20 21 22 23 24	<ol> <li>Section 2.6.1 on page 17, Section 16.2 on page 600 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.1.1 on page 789 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility.</li> </ol>
25 26 27 28 29 30 31 32 33 34	<ol> <li>Section 2.6.1 on page 17, Section 15.1 on page 594 and Section 16.1 on page 599. The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB, MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype MPI_Handler_function), MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, the dep- recated special datatype handles MPI_LB, MPI_UB, and the constants MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER, MPI_COMBINER_STRUCT_INTEGER were removed from the standard. This change may affect backward compatibility.</li> </ol>
35 36 37	3. Section 2.3 on page 10. Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.
38 39 40 41 42	4. Section 2.5.4 on page 15 and Section 7.5.4 on page 296. The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was in- troduced.
43 44 45 46	<ol> <li>Section 2.5.4 on page 15 and Section 8.1.1 on page 333. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.</li> </ol>
47 48	<ol> <li>Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 17, 25, 27, 176, Sections 4.1, 4.1.7, 4.1.8, 4.1.11, 12.3 on pages 83, 106, 107, 111, 480, and Annex A.1.1 on page 665.</li> </ol>

New inquiry functions, MPI\_TYPE\_SIZE\_X, MPI\_TYPE\_GET\_EXTENT\_X, 1  $\mathbf{2}$ MPI\_TYPE\_GET\_TRUE\_EXTENT\_X, and MPI\_GET\_ELEMENTS\_X, return their re-3 sults as an MPI\_Count value, which is a new type large enough to represent ele-4 ment counts in memory, file views, etc. A new function,  $\mathbf{5}$ MPI\_STATUS\_SET\_ELEMENTS\_X, modifies the opaque part of an MPI\_Status object so that a call to MPI\_GET\_ELEMENTS\_X returns the provided MPI\_Count value (in 6  $\overline{7}$ Fortran, INTEGER (KIND=MPI\_COUNT\_KIND). The corresponding predefined datatype is MPI\_COUNT. 8 9 7. Chapter 3 on page 23 until Chapter 17 on page 601. 10 In the C language bindings, the array-arguments' interfaces were modified to consis-11 tently use use [] instead of \*. 12Exceptions are MPI\_INIT, which continues to use char **\*\*\*argv** (correct because of 13 subtle rules regarding the use of the & operator with char \*argv[]), and 14MPI\_INIT\_THREAD, which is changed to be consistent with MPI\_INIT. 15168. Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 30, 101, 111, 130. 17 The functions MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS were defined to set the 18 count argument to MPI\_UNDEFINED when that argument would overflow. The func-19 tions MPI\_PACK\_SIZE and MPI\_TYPE\_SIZE were defined to set the size argument 20to MPI\_UNDEFINED when that argument would overflow. In all other MPI-2.2 rou-21tines, the type and semantics of the count arguments remain unchanged, i.e., int or 22 INTEGER. 23 $^{24}$ 9. Section 3.2.6 on page 32, and Section 3.8 on page 64. 25MPI\_STATUS\_IGNORE can be also used in MPI\_IPROBE, MPI\_PROBE, MPI\_IMPROBE, 26and MPI\_MPROBE. 2710. Section 3.8 on page 64 and Section 3.11 on page 80. 28The use of MPI\_PROC\_NULL in probe operations was clarified. A special predefined 29 message MPI\_MESSAGE\_NO\_PROC was defined for the use of matching probe (i.e., the 30 new MPI\_MPROBE and MPI\_IMPROBE) with MPI\_PROC\_NULL.  $^{31}$ 32 11. Sections 3.8.2, 3.8.3, 17.2.4, A.1.1 on pages 67, 69, 650, 665. 33 Like MPI\_PROBE and MPI\_IPROBE, the new MPI\_MPROBE and MPI\_IMPROBE 34 operations allow incoming messages to be queried without actually receiving them, 35 except that MPI\_MPROBE and MPI\_IMPROBE provide a mechanism to receive the 36 specific message with the new routines MPI\_MRECV and MPI\_IMRECV regardless of 37 other intervening probe or receive operations. The opaque object MPI\_Message, the 38 null handle MPI\_MESSAGE\_NULL, and the conversion functions MPI\_Message\_c2f and 39 MPI\_Message\_f2c were defined. 40 12. Section 4.1.2 on page 85 and Section 4.1.13 on page 115. 41 The routine MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK and constant 42MPI\_COMBINER\_HINDEXED\_BLOCK were added. 43 4413. Chapter 5 on page 141 and Section 5.12 on page 196. 45Added nonblocking interfaces to all collective operations. 464714. Sections 6.4.2, 6.4.4, 11.2.7, on pages 237, 248, 415. 48 The new routines MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO,

1 2 2		MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.
3 4 5	15.	Section 6.4.2 on page 237. Added MPI_COMM_IDUP.
6 7 8 9 10	16.	Section 6.4.2 on page 237. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.
11 12 13	17.	Section 6.4.2 on page 237. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- stant MPI_COMM_TYPE_SHARED.
14 15 16 17 18	18.	Section 6.6.2 on page 260. In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation could interfere with point-to-point communication on the parent communicator with the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0.
19 20 21 22	19.	Section 6.8 on page 281. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.
23 24 25	20.	Section 7.5.8 on page 312. MPI_CART_MAP can also be used for a zero-dimensional topologies.
23 26 27 28 29 30 31 32 33 34 35 36	21.	Section 7.6 on page 314 and Section 7.7 on page 323. The following neighborhood collective communication routines were added to support sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as address size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule was added for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT.
37 38 39 40 41	22.	Section 8.7 on page 355 and Section 12.4.3 on page 485. The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After MPI is initialized, the application can access information about the execution envi- ronment by querying the new predefined info object MPI_INFO_ENV.
42 43	23.	Section 8.7 on page 355. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.
44 45 46 47 48	24.	Chapter 11 on page 401. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes.

25.	Section 14.3 on page 563. A new MPI Tool Information Interface was added.	1 $2$
	The following changes are related to the Fortran language support.	3 4
26.	Section 2.3 on page 10, and Sections 17.1.1, 17.1.2, 17.1.7 on pages 601, 603, and 617. The new mpi_08 Fortran module was introduced.	5 6
27.	Section 2.5.1 on page 12, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 603, 605, and 617. Handles to opaque objects were defined as named types within the mpi_08 Fortran module. The operators .EQ., .NE., ==, and /= were overloaded to allow the comparison of these handles. The handle types and the overloaded operators are also available through the mpi Fortran module.	7 8 9 10 11 12
28.	Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 17.1.1, 17.1.10, 17.1.11, 17.1.12, 17.1.13 on pages 601, 628, 629, 630, 633, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 603, 605, 617. Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [41], and the compile-time constant MPI_SUBARRAYS_SUPPORTED was 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	13 14 15 16 17 18 19 20 21 22
29.	Section 2.6.2 on page 19, Section 17.1.2 on page 603, and Section 17.1.7 on page 617. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.	23 24
30.	Section 3.2.5 on page 30, Sections 17.1.2, 17.1.3, 17.1.7, on pages 603, 605, 617, and Section 17.2.5 on page 652. Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). 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. New conversion routines were 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 were added.	25 26 27 28 29 30 31 32 33 34
31.	Section 3.6 on page 44. In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument of MPI_BUFFER_DETACH is incorrectly defined and the argument is therefore unused.	35 36 37
32.	Section 4.1 on page 83, Section 4.1.6 on page 103, and Section 17.1.15 on page 633. The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in $C/C++$ ) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI_TYPE_CREATE_RESIZED to this datatype handle.	38 39 40 41 42 43 44 45
33.	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, 17.1.9 on pages 110, 183, 189, 275, 281, 341, 343, 345, 594, and 620. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords	46 47 48

1		TYPE and FUNCTION. The new dummy argument names must be used because the
2		mpi and mpi_08 modules guarantee keyword-based actual argument lists. The ar-
3		gument name type was changed in MPI_TYPE_DUP, the Fortran
4		USER_FUNCTION of MPI_OP_CREATE, MPI_TYPE_SET_ATTR,
5		MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME,
6		MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype defini-
7		tion MPI_Type_delete_attr_function, and the predefined callback function
8		MPI_TYPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE,
9		MPI_COMM_CREATE_ERRHANDLER, MPI_WIN_CREATE_ERRHANDLER,
10		MPI_FILE_CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consis-
11		
		tency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and
12		intracomm to newintracomm in MPI_INTERCOMM_MERGE.
13	9.4	
14	34.	Section 6.7.2 on page 267.
15		Section 6.7.2 on page 226. It was clarified that in Fortran, the flag values returned
16		by a $comm_copy_attr_fn$ callback, including MPI_COMM_NULL_COPY_FN and
17		MPI_COMM_DUP_FN, are .FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL.
18	35.	Section 8.2 on page 337.
19		With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports
20		TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may
21		be usable together with a non-standard Cray-pointer.
22		be usable together with a non-standard Cray-pointer.
23	36	Section $17.1.15$ on page $633$ , and Section $17.1.7$ on page $617$ .
	50.	
24		Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers
25		in MPI operations.
26	97	Section 17.1.16 on page 625 to Section 17.1.10 on page 646. Section 17.1.7 on page 617
27	57.	Section 17.1.16 on page 635 to Section 17.1.19 on page 646, Section 17.1.7 on page 617,
28		and Section 17.1.8 on page 619.
29		The sections about Fortran optimization problems and their solutions were partially
30		rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute.
31		The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the semantics of
		the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The For-
32		tran routine MPI_F_SYNC_REG is added. MPI-3.0 compliance for an MPI library
33		together with a Fortran compiler is defined in Section 17.1.7.
34		together with a Portran complet is defined in Section 17.1.7.
35	38	Section $17.1.2$ on page $603$ .
36	<b>J</b> 0.	* 0
37		Within the mpi_08 Fortran module, dummy arguments are now declared with
		INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.
38	20	Castion 17.1.2 on many COV and Castion 17.1.7 on many C17
39	ა9.	Section 17.1.3 on page 605, and Section 17.1.7 on page 617.
40		The existing mpi Fortran module must implement compile-time argument checking.
41	40	Section 17.1.4 on name 607
42	40.	Section 17.1.4 on page 607.
43		The use of the mpif.h Fortran include file is now strongly discouraged.
44	4 1	
	41.	Section A.1.1, Table "Predefined functions" on page 673, Section A.1.3 on page 680,
45		and Section A.3.4 on page 726.
46		Within the new mpi_f08 module, all callback prototype definitions are now defined
47		with explicit interfaces PROCEDURE(MPI) that have the BIND(C) attribute; user-
48		written callbacks must be modified if the mpi_f08 module is used.
		<b>▲</b> =

42.	Section A.1.3 on page 680.	1
	In some routines, the Fortran callback prototype names were changed fromFN to	2
	$\dots$ _FUNCTION to be consistent with the other language bindings.	3
		4
B.2	Changes from Version 2.1 to Version 2.2	5
D.2	Changes from version 2.1 to version 2.2	6
1	Section $2.5.4$ on page $15$ .	7
1.	It is now guaranteed that predefined named constant handles (as other constants)	8
	can be used in initialization expressions or assignments, i.e., also before the call to	9
	MPI_INIT.	10
		11
2.	Section $2.6$ on page $17$ , and Section $16.2$ on page $600$ .	12
	The C++ language bindings have been deprecated and may be removed in a future	13
	version of the MPI specification.	14
9	Castion 2.0.0 on page 25	15
э.	Section 3.2.2 on page 25. MPI_CHAR for printable characters is now defined for C type char (instead of signed	16
	char). This change should not have any impact on applications nor on MPI libraries	17
	(except some comment lines), because printable characters could and can be stored in	18
	any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed	19
	for predefined reduction operations.	20
	for predefined reduction operations.	21
4.	Section $3.2.2$ on page $25$ .	22
	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,	23
	MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and	24
	$MPI_C\_LONG\_DOUBLE\_COMPLEX$ are now valid predefined $MPI$ datatypes.	25
-		26
э.	Section 3.4 on page 37, Section 3.7.2 on page 48, Section 3.9 on page 73, and Section 5.1	27
	on page 141. The read access restriction on the cond buffer for blacking, non blacking and collective	28
	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	29
	-	30 31
	operation is in progress.	31
6.	Section 3.7 on page 47.	33
	The Advice to users for IBSEND and IRSEND was slightly changed.	34
_		35
7.	Section 3.7.3 on page 52.	36
	The advice to free an active request was removed in the Advice to users for	37
	MPI_REQUEST_FREE.	38
8.	Section $3.7.6$ on page $63$ .	39
	MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.	40
		41
9.	Section 5.8 on page 168.	42
	"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and	43
	MPI_ALLTOALLW for intracommunicators.	44
10	Section $5.9.2$ on page $176$ .	45
10.	Predefined parameterized datatypes (e.g., returned by	46
	MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g.	47
	MPI_REAL8) have been added to the list of valid datatypes in reduction operations.	48

1 2 3 4 5 6	11.	Section 5.9.2 on page 176. 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.
7 8 9 10	12.	Section 5.9.7 on page 189. The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.
11 12 13 14	13.	Section 5.10.1 on page 190. The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan- dard.
15 16	14.	Section 5.11.2 on page 194. Added in place argument to MPI_EXSCAN.
17 18 19 20 21 22 23	15.	Section 6.4.2 on page 237, and Section 6.6 on page 257. Implementations that did not implement MPI_COMM_CREATE on intercommuni- cators will need to add that functionality. As the standard described the behav- ior of this operation on intercommunicators, it is believed that most implementa- tions already provide this functionality. Note also that the C++ binding for both MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.
24 25 26 27 28	16.	Section 6.4.2 on page 237. MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intracommunicator. If comm is an intercommunicator it was clarified that all processes in the same local group of comm must specify the same value for group.
29 30 31 32 33	17.	Section 7.5.4 on page 296. New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ class Distgraphcomm were added.
34 35 36 37 38	18.	Section 7.5.5 on page 302. For the scalable distributed graph topology interface, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS and the constant MPI_DIST_GRAPH were added.
39 40 41 42	19.	Section 7.5.5 on page 302. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
43 44	20.	Section 8.1.1 on page 333. The subversion number changed from 1 to 2.
45 46 47 48	21.	Section 8.3 on page 340, Section 15.2 on page 597, and Annex A.1.3 on page 680. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.

22.	Section 8.7.1 on page 360. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.	1 2 3 4
23.	Section 11.3.4 on page 423. The restriction added in MPI 2.1 that the operation MPI_REPLACE in 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. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.	5 6 7 8 9 10 11 12
24.	Section 12.2 on page 473. Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.	13 14 15 16 17
25.	Section 13.7.2 on page 534, and Table 13.2 on page 536. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the external32 representation.	18 19 20 21 22
26.	Section 17.2.7 on page 657. The description was modified that it only describes how an MPI implementation behaves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples 17.13, 17.14, and 17.15 on pages 658-659 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.	23 24 25 26 27 28 29
27.	Annex A.1.1 on page 665. Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 679).	30 31 32
28.	Annex A.1.1 on page 665. Table Named Predefined Datatypes. Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.	33 34 35 36 37
B.3	Changes from Version 2.0 to Version 2.1	38 39
1.	Section 3.2.2 on page 25, and Annex A.1 on page 665. In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn- onym for MPI_LONG_LONG_INT.	40 41 42 43
2.	<ul> <li>Section 3.2.2 on page 25, and Annex A.1 on page 665.</li> <li>MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym),</li> <li>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.</li> </ul>	44 45 46 47 48

1 2 3 4 5 6	3.	Section 3.2.5 on page 30. 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.
7 8 9 10 11	4.	Section 4.1 on page 83. 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 datatype bounds or extent.
12 13 14 15	5.	Section 4.3 on page 137. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
16 17 18 19 20	6.	Section 5.9.6 on page 187. If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should provide count and datatype arguments that specify the same type signature (i.e., it is not necessary that both groups provide the same count value).
21 22 23 24	7.	Section 6.3.1 on page 228. 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 as the translated rank.
25 26 27 28 29 30 31 32 33 34 25	8.	Section 6.7 on page 265. About the attribute caching functions: <i>Advice to implementors.</i> High-quality implementations should raise an er- ror 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 key- val, information on the type of the associated user function. ( <i>End of advice to</i>
35		implementors.)
36 37 38 39 40	9.	Implementors.) Section 6.8 on page 281. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For- tran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT_NAME.
36 37 38 39		Section 6.8 on page 281. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For- tran, name is padded on the right with blank characters. resultlen cannot be larger

12.	Section 7.5.3 on page 294. In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$ , then MPI_COMM_NULL is returned in all processes.	1 2 3
13.	Section 7.5.3 on page 294. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.	4 5 7 8 9
	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> )	10 11 12 13
14.	Section 7.5.5 on page 302. 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.	14 15 16 17 18
15.	Section 7.5.5 on page 302. 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.	19 20 21
16.	Section 7.5.5 on page 302. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.	22 23 24 25
17.	Section 7.5.6 on page 310. 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.	26 27 28 29 30
18.	Section 7.5.7 on page 311. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.	31 32 33 34 35
18.1.	Section $8.1.1$ on page $333$ . The subversion number changed from 0 to 1.	36 37 38
19.	Section 8.1.2 on page 334. 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.	39 40 41 42 43
20.	Section 8.3 on page 340. 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	44 45 46 47 48

1 2 3	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.
4 5 2 6 7 8 9	<ol> <li>Section 8.7 on page 355, 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 collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 397.</li> </ol>
10 11 12	2. Section 8.7 on page 355. About MPI_ABORT:
13 14 15 16	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. ( <i>End of advice to users.</i> )
17 18 19	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.)
20 21 22 23 24 25 26 27 28	3. Section 9 on page 365. 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.
29 30 31 32 33	4. Section 11.3 on page 416. 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 point- to-point communication. See also item 25 in this list.
34 2 35 36 37 38	5. Section 11.3 on page 416. 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.
	<ol> <li>Section 11.3.4 on page 423.</li> <li>MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.</li> </ol>
42 43 44 45 46 47 48	7. Section 13.2.8 on page 498. 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 498. 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.	1 2 3
29.	Section 13.3 on page 501. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.	4 5 6 7
30.	Section 13.7.2 on page 534. The bias of 16 byte doubles was defined with 10383. The correct value is 16383.	8 9 10
31.	MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0). In the example in this section, the buffer should be declared as const void* buf.	11 12
32.	Section 17.1.9 on page 620. About MPI_TYPE_CREATE_F90_XXX:	13 14 15
	Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of $(XXX,p,r)$ . 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_XXX and using a hash- table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (End of advice to implementors.)	16 17 18 19 20 21 22 23 24 25 26
33.	Section A.1.1 on page 665. MPI_BOTTOM is defined as void * const MPI::BOTTOM.	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

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] 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

1 2 3	[12]	S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communica- tion performance of large multicomputers. In <i>Proceedings of the 1991 International</i> <i>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]	Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings, volume 6960 of Lecture Notes in Computer Science. Springer, 2011. 16, 36
12 13 14 15	[15]	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
16 17 18 19	[16]	James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. In Cotronis et al. [14], pages 282–291. 6.4.2
20 21 22	[17]	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
23 24 25 26	[18]	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
27 28 29	[19]	Edinburgh Parallel Computing Centre, University of Edinburgh. <i>CHIMP Concepts</i> , June 1991. 1.2
23 30 31	[20]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
32 33 34 35	[21]	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
36 37 38 39	[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
40 41	[23]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3
42 43 44 45	[24]	Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. <i>PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.</i> MIT Press, 1994. 10.1
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. 17.1.3
- [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. 3.8.1, 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] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI common practice, issues, and extensions. In Cotronis et al. [14], pages 345–355. 6.4.2
- [37] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.7.2
- [38] International Organization for Standardization, Geneva, ISO 8859-1:1987. Information 46 processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet 47 No. 1, 1987. 13.7.2

805

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

1 2 3	[39]	International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). In- formation technology — Portable Operating System Interface (POSIX) — Part 1: Sys- tem Application Program Interface (API) [C Language], December 1996. 12.4, 13.2.1
4 5 6 7	[40]	International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. In- formation technology – Programming languages – Fortran – Part 1: Base language, November 2010. 17.1.1, 17.1.2
8 9 10 11 12 13	[41]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TS 29113. TS on further interoperability with C, 2012. http://www.nag.co.uk/sc22wg5/, successfully balloted DTS at ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf. 17.1.1, 17.1.1, 17.1.2, 17.1.7, 28
14 15	[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
16 17 18 19	[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
20 21 22	[44]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 7.1
23 24 25 26 27	[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
28 29	[46]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2
30 31 32	[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
33 34	[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
35 36 37	[49]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 10.5.5
38 39 40 41	[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
42 43 44	[51]	Martin Schulz and Bronis R. de Supinski. $P^NMPI$ 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
45 46 47 48	[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

- [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, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. 1.2, 6.1.2
- [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
- [55] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. 6.1
- [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
- [57] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994. 6.1.2, 6.5.6
- [58] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996. 13.1
- [59] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.7.2
- [60] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

 $\overline{7}$ 

 $^{24}$ 

#### **Examples Index**

10

11

12 13 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[ticket281.][/C++] examples.

14	ASYNCHRONOUS, 639, 647
15	Attributes between languages, 658
16	Basic tool using performance variables in the
17	MPI tool information interface, 584
18	
19	C/Fortran handle conversion, $651$
20	Cartesian virtual topologies, 329
21	Client-server code, 62, 63
22	with blocking probe, 66
23	with blocking probe, wrong, $66$
24	Datatype
25	3D array, $121$
26	absolute addresses, 127
27	array of structures, $124$
28	elaborate example, 134, 136
29	matching type, 111
30	matrix transpose, 123
	union, 128
31	Datatypes matching 23
32	matching, 33 not matching, 34
33	untyped, 34
34	Deadlock
35	if not buffered, $43$
36	with MPI_Bcast, 214, 215
37	wrong message exchange, $43$
38	
39	False matching of collective operations, 218
40	Fortran 90 copying and sequence problem, 630–632
41	Fortran 90 derived types, 634
42	Fortran 90 heterogeneous communication, 627
43	Fortran 90 invalid KIND, 622
44	Fortran 90 MPI_TYPE_MATCH_SIZE
45	implementation, 626
46	Fortran 90 overlapping communication and
47	computation, $644, 645, 648$
48	Fortran 90 register optimization, 637, 638

Independence of nonblocking operations, 221 Intercommunicator, 241, 245 Interlanguage communication, 662 Intertwined matching pairs, 41

Message exchange, 42Mixing blocking and nonblocking collective operations, 217 Mixing collective and point-to-point requests, 220MPI\_ACCUMULATE, 425 MPI\_Accumulate, 466, 467, 469 MPI\_Aint, 124 MPI\_Allgather, 167 MPI\_ALLOC\_MEM, 339 MPI\_Alloc\_mem, 339, 469 MPI\_ALLREDUCE, 188 MPI\_Alltoall, 219 MPI\_Barrier, 358, 456-459, 465-468 MPI\_Bcast, 149, 214-218 MPI\_BSEND, 41 MPI\_Buffer\_attach, 45, 358 MPI\_Buffer\_detach, 45 MPI\_BYTE, 34 MPI\_Cancel, 358 MPI\_CART\_COORDS, 311 MPI\_CART\_GET, 329 MPI\_CART\_RANK, 311 MPI\_CART\_SHIFT, 311, 329  $\mathrm{MPI\_CART\_SUB},\, \frac{312}{}$ MPI\_CHARACTER, 35 MPI\_Comm\_create, 241, 251, 252, 255 MPI\_Comm\_create\_keyval, 279 MPI\_Comm\_dup, 254 MPI\_Comm\_get\_attr, 279 MPI\_Comm\_group, 241, 255, 279 MPI\_Comm\_remote\_size, 245 MPI\_Comm\_set\_attr, 279 MPI\_COMM\_SPAWN, 376 MPI\_Comm\_spawn, 376

MPI\_COMM\_SPAWN\_MULTIPLE, 382 MPI\_Comm\_spawn\_multiple, 382 MPI\_Comm\_split, 245, 263, 264 MPI\_Compare\_and\_swap, 468, 469 MPI\_DIMS\_CREATE, 293, 329 MPI\_DIST\_GRAPH\_CREATE, 300 MPI\_Dist\_graph\_create, 301 MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, 300 MPI\_FILE\_CLOSE, 512, 515 MPI\_FILE\_GET\_AMODE, 497 MPI\_FILE\_IREAD, 515 MPI\_FILE\_OPEN, 512, 515 MPI\_FILE\_READ, 512 MPI\_FILE\_SET\_ATOMICITY, 546 MPI\_FILE\_SET\_VIEW, 512, 515 MPI\_FILE\_SYNC, 547 MPI\_Finalize, 357-359 MPI\_FREE\_MEM, 339 MPI\_Free\_mem, 469 MPI\_Gather, 136, 152, 153, 157 MPI\_Gatherv, 136, 154–157 MPI\_GET, 421, 422 MPI\_Get, 456–459, 464, 465 MPI\_Get\_accumulate, 466, 467, 469 MPI\_GET\_ADDRESS, 102, 634, 635, 655 MPI\_Get\_address, 124, 127, 128, 134 MPI\_GET\_COUNT, 113 MPI\_GET\_ELEMENTS, 113 MPI\_GRAPH\_CREATE, 294, 307 MPI\_GRAPH\_NEIGHBORS, 307 MPI\_GRAPH\_NEIGHBORS\_COUNT, 307 MPI\_Grequest\_complete, 478 MPI\_Grequest\_start, 478 MPI\_Group\_excl, 251 MPI\_Group\_free, 241, 251, 252 MPI\_Group\_incl, 241, 252, 255 MPI\_Iallreduce, 220 MPI\_Ialltoall, 219 MPI\_Ibarrier, 217–220 MPI\_Ibcast, 199, 220, 221 MPI\_INFO\_ENV, 356 MPI\_Intercomm\_create, 263, 264 MPI\_Iprobe, 358 MPI\_IRECV, 54-56, 62, 63 MPI\_Irecv, 220 MPI\_ISEND, 54-56, 62, 63 MPI\_Op\_create, 186, 187, 195 MPI\_Pack, 134, 136 MPI\_Pack\_size, 136 MPI\_PROBE, 66 MPI\_Put, 442, 447, 457-459, 464, 465 MPI\_RECV, 33–35, 41–43, 56, 66, 111 MPI\_Recv, 219

MPI_REDUCE, 177, 178, 181	1
MPI_Reduce, 180, 182, 186, 187	2
MPI_REQUEST_FREE, 55	3
MPI_Request_free, 358	4
$MPI_Rget, 468$	5
MPI_Rput, 468	
MPI_Scan, 195	6
MPI_Scatter, 162	7
MPI_Scatterv, 162, 163	8
MPI_SEND, 33–35, 42, 43, 56, 66, 111	9
MPI_Send, 124, 127, 128, 134, 219, 220	10
MPI_SENDRECV, 121–123	11
MPI_SENDRECV_REPLACE, 311	
MPI_SSEND, 41, 56	12
MPI_Test_cancelled, 358	13
MPI_TYPE_COMMIT, 109, 121–123, 421,	14
634, 635	15
MPI_Type_commit, 124, 127, 128, 134,	16
153–157, 163, 195	17
MPI_TYPE_CONTIGUOUS, 86, 104, 111, 113	18
MPI_Type_contiguous, 153	
MPI_TYPE_CREATE_DARRAY, 101	19
MPI_TYPE_CREATE_HVECTOR, 121, 123	20
MPI_Type_create_hvector, 124, 127	21
MPI_Type_create_invector, 124, 127 MPI_TYPE_CREATE_INDEXED_BLOCK,	22
	23
421 MDL TYPE CDEATE DECIZED COA COT	24
MPI_TYPE_CREATE_RESIZED, 634, 635	
MPI_TYPE_CREATE_STRUCT, 93, 104,	25
123, 634, 635	26
MPI_Type_create_struct, 124, 127, 128, 134,	27
156, 157, 195	28
MPI_TYPE_CREATE_SUBARRAY, 553	29
MPI_TYPE_EXTENT, 421	30
MPI_TYPE_FREE, 421	
MPI_Type_get_contents, 129	31
MPI_Type_get_envelope, 129	32
MPI_TYPE_GET_EXTENT, 121, 123, 422,	33
425	34
$MPI_Type_get_extent, 124$	35
MPI_TYPE_INDEXED, 89, 122	36
MPI_Type_indexed, 124, 127	
MPI_TYPE_VECTOR, 87, 121, 123	37
MPI_Type_vector, 154, 155, 157, 163	38
MPI_Unpack, 134, 136	39
MPI_User_function, 187	40
MPI_WAIT, 54–56, 62, 63, 515	41
MPI_Wait, 217–220	42
MPI_Waitall, 220, 468	43
MPI_WAITANY, 62	
MPI_Waitany, 468	44
MPI_WAITSOME, 63	45
MPI_Win_attach, 469	46
MPI_Win_complete, 442, 459, 465	47
MPI_WIN_CREATE, 421, 422, 425	48
, , , ,	

```
1
      MPI_Win_create_dynamic, 469
^{2}
      MPI_Win_detach, 469
      MPI_WIN_FENCE, 421, 422, 425
3
      MPI_Win_fence, 464
4
      MPI_Win_flush, 458, 466-469
\mathbf{5}
      MPI_Win_flush_all, 467
6
      MPI_Win_flush_local, 457
\overline{7}
      MPI_WIN_FREE, 422, 425
8
      MPI_Win_lock, 447, 456-459
9
      MPI_Win_lock_all, 468, 469
      MPI_Win_post, 459, 465
10
      MPI_Win_start, 442, 459, 465
11
      MPI_Win_sync, 457, 458, 466-468
12
      MPI_Win_unlock, 447, 456-459
13
      MPI_Win_unlock_all, 468, 469
14
      MPI_Win_wait, 459, 465
15
      mpiexec, 356, 363
16
      Neighborhood collective communication, 329
17
      No Matching of Blocking and Nonblocking
18
               collective operations, 219
19
      Non-deterministic program with MPI_Bcast,
20
               216
21
      Non-overtaking messages, 41
22
      Nonblocking operations, 54, 55
23
           message ordering, 56
           progress, 56
^{24}
25
      Overlapping Communicators, 220
26
27
      Pipelining nonblocking collective operations,
28
               220
      Profiling interface, 560
29
      Progression of nonblocking collective
30
               operations, 219
31
32
      Reading the value of a control variable in the
33
               MPI tool information interface, 574
34
      Threads and MPI, 483
35
      Topologies, 329
36
      Typemap, 85-87, 89, 93, 101
37
38
      Using MPI_T_CVAR_GET_INFO to list all
39
               names of control variables., 571
40
      Virtual topologies, 329
41
42
43
44
45
46
47
48
```

### MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI::\_LONG\_LONG, 790 MPI::BOOL, 789 MPI::COMPLEX, 789 MPI::DOUBLE\_COMPLEX, 789 MPI::F\_COMPLEX16, 790 MPI::F\_COMPLEX32, 790 MPI::F\_COMPLEX4, 790 MPI::F\_COMPLEX8, 790 MPI::INTEGER16, 790 MPI::LONG\_DOUBLE\_COMPLEX, 789 MPI::LONG\_LONG, 790 MPI::REAL16, 790 MPI\_2DOUBLE\_PRECISION, 180, 671 MPI\_2INT, 180, 671 MPI\_2INTEGER, 180, 671 MPI\_2REAL, 180, 671 MPI\_ADDRESS\_KIND, 15, 16, 16, 26, 266, 629, 658, 668 MPI\_AINT, 25, 27, 177, 411, 669, 670, 795-797 MPI\_ANY\_SOURCE, 28, 29, 41, 51, 52, 64, 65, 67-69, 76, 79, 80, 287, 335, 667 MPI\_ANY\_TAG, 15, 28, 29, 31, 51, 52, 64, 65, 67-71, 76, 79-81, 667, 792 MPI\_APPNUM, 396, 397, 674 MPI\_ARGV\_NULL, 16, 376, 377, 628, 676 MPI\_ARGVS\_NULL, 16, 381, 628, 676 MPI\_ASYNC\_PROTECTS\_NONBLOCKING, 15, 602, 603, 606, 608, 609, 618, 620, 640, 667, 794 MPI\_BAND, 176, 177, 672 MPI\_BOR, 176, 177, 672 MPI\_BOTTOM, 10, 16, 32, 101, 114, 115, 145, 298, 300, 378, 411, 414, 604, 607, 614, 628, 633, 635, 637-639, 641-643, 646, 655, 656, 662, 667, 801 MPI\_BSEND\_OVERHEAD, 46, 667 MPI\_BXOR, 176, 177, 672 MPI\_BYTE, 25, 26, 33, 34, 36, 137, 177, 490, 532, 544, 662, 669, 670, 798 MPI\_C\_BOOL, 26, 177, 669, 790, 795–797

MPI_C_COMPLEX, 26, 177, 669, 789,	13
795–797	14
MPI_C_DOUBLE_COMPLEX, 26, 177, 669,	15
795–797	16
MPI_C_FLOAT_COMPLEX, 177, 669,	17
795–797	18
MPI_C_LONG_DOUBLE_COMPLEX, 26,	19
177,669,795797	20
MPI_CART, 302, 673	
MPI_CHAR, 26, 36, 93, 178, 179, 567, 669, 795	21
MPI_CHARACTER, 25, 34–36, 178, 179, 670	22
MPI_COMBINER_CONTIGUOUS, 116, 119,	23
675	24
MPI_COMBINER_DARRAY, 116, 121, 675	25
MPI_COMBINER_DUP, 116, 119, 675	26
MPI_COMBINER_F90_COMPLEX, 116, 121,	27
675	28
MPI_COMBINER_F90_INTEGER, 116, 121,	29
675	30
MPI_COMBINER_F90_REAL, 116, 121, 675	31
MPI_COMBINER_HINDEXED, 18, 116, 120, 675	
MPI_COMBINER_HINDEXED_BLOCK, 116,	32
120, 675, 791	33
MPI_COMBINER_HINDEXED_INTEGER,	34
18, 600, 790	35
MPI_COMBINER_HVECTOR, 18, 116, 120,	36
675	37
MPI_COMBINER_HVECTOR_INTEGER,	38
18, 600, 790	39
MPI_COMBINER_INDEXED, 116, 120, 675	40
MPI_COMBINER_INDEXED_BLOCK, 116,	41
120, 675	
MPI_COMBINER_NAMED, 116, 117, 119,	42
675	43
MPI_COMBINER_RESIZED, 116, 121, 675	44
MPI_COMBINER_STRUCT, 18, 116, 120, 675	45
MPI_COMBINER_STRUCT_INTEGER, 18,	46
600, 790	47
MPI_COMBINER_SUBARRAY, 116, 121, 675	48

> 5 6

1 MPI\_COMBINER\_VECTOR, 116, 120, 675  $\mathbf{2}$ MPI\_COMM\_DUP\_FN, 18, 269, 612, 673, 794 MPI\_COMM\_NULL, 227, 240, 241, 243-245, 3 247, 248, 283, 292, 294, 379, 398-400, 4 672.799  $\mathbf{5}$ MPI\_COMM\_NULL\_COPY\_FN, 18, 269, 604, 6 656, 673, 794 7MPI\_COMM\_NULL\_DELETE\_FN, 18, 269, 8 673 9 MPI\_COMM\_PARENT, 283 10 MPI\_COMM\_SELF, 227, 243, 266, 283, 360, 361, 398, 492, 671, 797 11MPI\_COMM\_TYPE\_SHARED, 248, 671, 792 12MPI\_COMM\_WORLD, 15, 22, 27, 28, 227, 13229, 236, 237, 252, 261, 283, 293, 14334-336, 340, 342, 351, 357, 359, 360, 15362, 371, 372, 374, 375, 379, 381, 16395-398, 486, 530, 550, 551, 573, 580, 17650, 661, 671, 800 MPI\_COMPLEX, 25, 177, 534, 620, 670 18 MPI\_COMPLEX16, 177, 670 19MPI\_COMPLEX32, 177, 670 20MPI\_COMPLEX4, 177, 670 21MPI\_COMPLEX8, 177, 670  $^{22}$ MPI\_CONGRUENT, 237, 259, 671 23MPI\_CONVERSION\_FN\_NULL, 539, 673  $^{24}$ MPI\_COUNT, 25, 27, 177, 567, 669, 670, 791 25MPI\_COUNT\_KIND, 15, 26, 668 MPI\_CXX\_BOOL, 27, 177, 670, 789 26MPI\_CXX\_DOUBLE\_COMPLEX, 27, 177, 27670, 789 28MPI\_CXX\_FLOAT\_COMPLEX, 27, 177, 670, 29789 30MPI\_CXX\_LONG\_DOUBLE\_COMPLEX, 27,  $^{31}$ 177, 670, 789 MPI\_DATATYPE\_NULL, 110, 672 32 MPI\_DISPLACEMENT\_CURRENT, 502, 33 676, 801 34MPI\_DIST\_GRAPH, 302, 673, 796 35 MPI\_DISTRIBUTE\_BLOCK, 98, 676 36 MPI\_DISTRIBUTE\_CYCLIC, 98, 676 37 MPI\_DISTRIBUTE\_DFLT\_DARG, 98, 676 38MPI\_DISTRIBUTE\_NONE, 98, 676 39 MPI\_DOUBLE, 26, 176, 567, 576, 577, 620, 40669 MPI\_DOUBLE\_COMPLEX, 25, 177, 534, 620,  $^{41}$ 670 42MPI\_DOUBLE\_INT, 180, 671 43MPI\_DOUBLE\_PRECISION, 25, 176, 620, 44670 45MPI\_DUP\_FN, 18, 269, 595, 674 46MPI\_ERR\_ACCESS, 349, 495, 551, 666 47MPI\_ERR\_AMODE, 349, 493, 551, 666 MPI\_ERR\_ARG, 348, 665 48

MPI\_ERR\_ASSERT, 348, 452, 666 MPI\_ERR\_BAD\_FILE, 349, 551, 666 MPI\_ERR\_BASE, 338, 348, 452, 666 MPI ERR BUFFER, 348, 665 MPI\_ERR\_COMM, 348, 665 MPI\_ERR\_CONVERSION, 349, 540, 551, 666 MPI\_ERR\_COUNT, 348, 665 MPI\_ERR\_DIMS, 348, 665 MPI\_ERR\_DISP, 348, 452, 666 MPI\_ERR\_DUP\_DATAREP, 349, 537, 551, 666 MPI\_ERR\_FILE, 349, 551, 666 MPI\_ERR\_FILE\_EXISTS, 349, 551, 666 MPI\_ERR\_FILE\_IN\_USE, 349, 495, 551, 666 MPI\_ERR\_GROUP, 348, 665 MPI\_ERR\_IN\_STATUS, 30, 32, 53, 59, 61, 342, 348, 477, 507, 666 MPI\_ERR\_INFO, 348, 666 MPI\_ERR\_INFO\_KEY, 348, 366, 666 MPI\_ERR\_INFO\_NOKEY, 348, 367, 666 MPI\_ERR\_INFO\_VALUE, 348, 366, 666 MPI\_ERR\_INTERN, 348, 665 MPI\_ERR\_IO, 349, 551, 666 MPI\_ERR\_KEYVAL, 279, 348, 666 MPI\_ERR\_LASTCODE, 347, 349, 351, 352, 590,667 MPI\_ERR\_LOCKTYPE, 348, 452, 666 MPI\_ERR\_NAME, 348, 392, 666 MPI\_ERR\_NO\_MEM, 338, 348, 666 MPI\_ERR\_NO\_SPACE, 349, 551, 666 MPI\_ERR\_NO\_SUCH\_FILE, 349, 494, 551, 666 MPI\_ERR\_NOT\_SAME, 349, 551, 666 MPI\_ERR\_OP, 348, 451, 665 MPI\_ERR\_OTHER, 347, 348, 665 MPI\_ERR\_PENDING, 59, 348, 665 MPI\_ERR\_PORT, 348, 389, 666 MPI\_ERR\_QUOTA, 349, 551, 666 MPI\_ERR\_RANK, 348, 451, 665 MPI\_ERR\_READ\_ONLY, 349, 551, 666 MPI\_ERR\_REQUEST, 348, 665 MPI\_ERR\_RMA\_ATTACH, 349, 452, 666 MPI\_ERR\_RMA\_CONFLICT, 348, 452, 666 MPI\_ERR\_RMA\_FLAVOR, 349, 409, 452, 666 MPI\_ERR\_RMA\_RANGE, 349, 452, 666 MPI\_ERR\_RMA\_SHARED, 349, 452, 666 MPI\_ERR\_RMA\_SYNC, 348, 452, 666 MPI\_ERR\_ROOT, 348, 665 MPI\_ERR\_SERVICE, 348, 391, 666 MPI\_ERR\_SIZE, 348, 452, 666 MPI\_ERR\_SPAWN, 348, 377, 378, 666 MPI\_ERR\_TAG, 348, 665 MPI\_ERR\_TOPOLOGY, 348, 665 MPI\_ERR\_TRUNCATE, 348, 665

MPI\_ERR\_TYPE, 348, 665 MPI\_ERR\_UNKNOWN, 347, 348, 665 MPI\_ERR\_UNSUPPORTED\_DATAREP, 349, 551,666 MPI\_ERR\_UNSUPPORTED\_OPERATION, 349, 551, 666 MPI\_ERR\_WIN, 348, 452, 666 MPI\_ERRCODES\_IGNORE, 16, 378, 628, 676 MPI\_ERRHANDLER\_NULL, 346, 672 MPI\_ERROR, 30, 53, 197, 430, 668, 793 MPI\_ERRORS\_ARE\_FATAL, 340, 341, 352, 353, 451, 550, 668 MPI\_ERRORS\_RETURN, 340, 341, 353, 550, 661, 668 MPI\_F08\_STATUS\_IGNORE, 653, 677, 793 MPI\_F08\_STATUSES\_IGNORE, 653, 677, 793 MPI\_F\_STATUS\_IGNORE, 652, 677 MPI\_F\_STATUSES\_IGNORE, 652, 677 MPI\_FILE\_NULL, 494, 550, 551, 672 MPI\_FLOAT, 26, 93, 174, 176, 533, 669 MPI\_FLOAT\_INT, 12, 180, 671 MPI\_GRAPH, 302, 673 MPI\_GROUP\_EMPTY, 226, 232, 240, 241, 243, 672 MPI\_GROUP\_NULL, 226, 235, 672 MPI\_HOST, 335, 671 MPI\_IDENT, 229, 237, 671 MPI\_IN\_PLACE, 16, 144, 171, 607, 628, 667 MPI\_INFO\_ENV, 356, 671, 792 MPI\_INFO\_NULL, 300, 370, 378, 387, 493,  $494,\,503,\,672$ MPI\_INT, 12, 26, 84, 176, 533, 534, 567, 570, 575, 579, 620, 661, 663, 669 MPI\_INT16\_T, 26, 176, 669, 795–797 MPI\_INT32\_T, 26, 176, 669, 795–797 MPI\_INT64\_T, 26, 176, 669, 795-797 MPI\_INT8\_T, 26, 176, 669, 795–797 MPI\_INTEGER, 25, 33, 176, 620, 663, 670 MPI\_INTEGER1, 25, 176, 670 MPI\_INTEGER16, 176, 670 MPI\_INTEGER2, 25, 176, 534, 670 MPI\_INTEGER4, 25, 176, 670 MPI\_INTEGER8, 176, 624, 670 MPI\_INTEGER\_KIND, 15, 668 MPI\_IO, 335, 671 MPI\_KEYVAL\_INVALID, 270, 271, 667 MPI\_LAND, 176, 177, 672 MPI\_LASTUSEDCODE, 351, 674 MPI\_LB, 18, 600, 790 MPI\_LOCK\_EXCLUSIVE, 445, 667 MPI\_LOCK\_SHARED, 445, 446, 667 MPI\_LOGICAL, 25, 177, 670 MPI\_LONG, 26, 176, 669 MPI\_LONG\_DOUBLE, 26, 176, 669

MPI_LONG_DOUBLE_INT, 180, 671	1
MPI_LONG_INT, 180, 671	2
MPI_LONG_LONG, 26, 176, 669, 797	3
MPI_LONG_LONG_INT, 26, 176, 669, 797	4
MPI_LOR, 176, 177, 672	
MPI_LXOR, 176, 177, 672	5
MPI_MAX, 174, 176, 177, 194, 672	6
MPI_MAX_DATAREP_STRING, 15, 504,	7
537, 668	8
MPI_MAX_ERROR_STRING, 15, 347, 352,	9
668	10
MPI_MAX_INFO_KEY, 15, 348, 365, 368, 668	11
MPI_MAX_INFO_VAL, 15, 348, 365, 668	
MPI_MAX_LIBRARY_VERSION_STRING,	12
15, 334, 668, 790	13
MPI_MAX_OBJECT_NAME, 15, 282–284,	14
	15
668, 792, 798	16
MPI_MAX_PORT_NAME, 15, 387, 668	
MPI_MAX_PROCESSOR_NAME, 15, 336,	17
337, 668, 799	18
MPI_MAXLOC, 176, 179, 180, 183, 672	19
MPI_MESSAGE_NO_PROC, 69–71, 667, 791	20
MPI_MESSAGE_NULL, 69–71, 672, 791	21
MPI_MIN, 176, 177, 672	22
MPI_MINLOC, 176, 179, 180, 183, 672	
MPI_MODE_APPEND, 492, 493, 675	23
MPI_MODE_CREATE, 492, 493, 501, 675	24
MPI_MODE_DELETE_ON_CLOSE, 492–494,	25
675	26
MPI_MODE_EXCL, 492, 493, 675	27
MPI_MODE_NOCHECK, 446, 450, 451, 675	28
MPI_MODE_NOPRECEDE, 440, 450, 451,	
675	29
MPI_MODE_NOPUT, 450, 451, 675	30
MPI_MODE_NOSTORE, 450, 451, 675	31
MPI_MODE_NOSUCCEED, 450, 451, 675	32
MPI_MODE_RDONLY, 492, 493, 498, 675	33
MPI_MODE_RDWR, 492, 493, 675	34
MPI_MODE_SEQUENTIAL, 492, 493, 495,	
496, 502, 507, 511, 522, 543, 675, 801	35
MPI_MODE_UNIQUE_OPEN, 492, 493, 675	36
MPI_MODE_WRONLY, 492, 493, 675	37
MPI_NO_OP, 427, 428, 672	38
MPI_NULL_COPY_FN, 18, 269, 595, 674	39
MPI_NULL_DELETE_FN, 18, 269, <u>595</u> , 674	40
MPI_OFFSET, 25, 177, 669, 670, 795–797	
MPI_OFFSET_KIND, 15, <u>17</u> , 26, 545, 629, 668	41
MPI_OP_NULL, 186, 672	42
MPI_ORDER_C, 15, 95, 98, 99, 676	43
MPI_ORDER_FORTRAN, 15, 95, 98, 99, 676	44
	45
MPI_PACKED, 12, 25, 26, 33, 131, 133, 137,	46
535, 662, 669, 670 MBL PROC NULL 24, 27, 20, 20, 65, 60, 71	47
MPI_PROC_NULL, 24, 27, 29, 30, 65, 69–71,	
80, 81, 146, 148, 150, 152, 160, 162,	48

1	175, 229, 311, 314, 335, 409, 417, 667,
2	791, 798, 800
	MPI_PROD, 176, 177, 672
3	
4	MPI_REAL, 25, 33, 176, 534, 620, 621, 627,
5	670
	MPI_REAL16, 177, 670
6	MPI_REAL2, 25, 177, 670
7	MPI_REAL4, 25, 177, 620, 624, 670
8	
	MPI_REAL8, 25, 177, 620, 670, 795
9	MPI_REPLACE, 425–428, 468, 672, 797, 800
10	MPI_REQUEST_NULL, 52–55, 57–61, 476,
11	672
	MPI_ROOT, 146, 667
12	
13	MPI_SEEK_CUR, 517, 523, 676
14	MPI_SEEK_END, 517, 523, 676
14	MPI_SEEK_SET, 517, 523, 676
15	MPI_SHORT, 26, 176, 669
16	MPI_SHORT_INT, 180, 671
17	MPI_SIGNED_CHAR, 26, 176, 178, 179, 669,
11	
18	797
19	MPI_SIMILAR, 229, 237, 259, 671
20	MPI_SOURCE, 30, 197, 668, 793
	MPI_STATUS_IGNORE, 10, 16, 32, 475, 507,
21	607, 628, 652, 653, 662, 676, 677, 791
22	MPI_STATUS_SIZE, 15, 30, 608, 668, 793
23	
	MPI_STATUSES_IGNORE, 14, 16, 32, 475,
24	477,628,652,653,676,677
25	MPI_SUBARRAYS_SUPPORTED, 15, 602,
26	603, 606, 608-611, 614, 616-618, 630,
~-	631, 667, 793
27	MPI_SUBVERSION, 15, 334, 677
28	
29	MPI_SUCCESS, 19, 52, 59, 61, 269, 271–274,
20	276, 277, 347, 348, 352, 353, 378, 540,
30	563,581583,590,591,595,665
31	MPI_SUM, 176, 177, 661, 672
32	MPI_T_BIND_MPI_COMM, 565, 678
22	MPI_T_BIND_MPI_DATATYPE, 565, 678
33	
34	MPI_T_BIND_MPI_ERRHANDLER, 565,
35	678
36	MPI_T_BIND_MPI_FILE, 565, 678
30	MPI_T_BIND_MPI_GROUP, 565, 678
37	MPI_T_BIND_MPI_INFO, 565, 678
38	MPI_T_BIND_MPI_MESSAGE, 565, 678
39	
33	MPI_T_BIND_MPI_OP, 565, 678
40	MPI_T_BIND_MPI_REQUEST, 565, 678
41	MPI_T_BIND_MPI_WIN, 565, 678
42	MPI_T_BIND_NO_OBJECT, 565, 571, 573,
	579-581, 678
43	MPI_T_CVAR_HANDLE_NULL, 573, 677
44	
45	MPI_T_CVAR_READ, WRITE, 591
	MPI_T_ENUM_NULL, 570, 579, 677
46	MPI_T_ERR_CANNOT_INIT, 591, 667
47	MPI_T_ERR_CVAR_SET_NEVER, 574, 591,
48	667
10	

MPI_T_ERR_CVAR_SET_NOT_NOW, 574, 591, 667
MPI_T_ERR_INVALID_HANDLE, 591, 667
MPI_T_ERR_INVALID_INDEX, 591, 667
MPI_T_ERR_INVALID_ITEM, 591, 667
MPI_T_ERR_INVALID_SESSION, 591, 667
MPI_T_ERR_MEMORY, 591, 667
MPI_T_ERR_NOT_INITIALIZED, 591, 667
MPI_T_ERR_OUT_OF_HANDLES, 591, 667
MPI_T_ERR_OUT_OF_SESSIONS, 591, 667
MPI_T_ERR_PVAR_NO_ATOMIC, 583, 591,
667 MPI_T_ERR_PVAR_NO_STARTSTOP, 581,
582, 591, 667
MPI_T_ERR_PVAR_NO_WRITE, 583, 591, 667
MPI_T_PVAR_ALL_HANDLES, 581–584, 678
MPI_T_PVAR_CLASS_AGGREGATE, 577, 678
MPI_T_PVAR_CLASS_COUNTER, 576, 678
MPI_T_PVAR_CLASS_GENERIC, 577, 678
MPI_T_PVAR_CLASS_HIGHWATERMARK,
576, 678
MPI_T_PVAR_CLASS_LEVEL, 576, 678
MPI_T_PVAR_CLASS_LOWWATERMARK,
576, 678 MPI_T_PVAR_CLASS_PERCENTAGE, 576,
678
678 MPI_T_PVAR_CLASS_SIZE, 576, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_HANDLE_NULL, 581, 677
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_HANDLE_NULL, 581, 677 MPI_T_PVAR_SESSION_NULL, 580, 677
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_HANDLE_NULL, 581, 677 MPI_T_PVAR_SESSION_NULL, 580, 677 MPI_T_SCOPE_ALL, 571, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_HANDLE_NULL, 581, 677 MPI_T_PVAR_SESSION_NULL, 580, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_HANDLE_NULL, 581, 677 MPI_T_PVAR_SESSION_NULL, 580, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678 MPI_T_SCOPE_CONSTANT, 571, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_HANDLE_NULL, 581, 677 MPI_T_PVAR_SESSION_NULL, 580, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678 MPI_T_SCOPE_CONSTANT, 571, 678 MPI_T_SCOPE_GROUP, 571, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_STATE, 577, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_BESSION_NULL, 581, 677 MPI_T_PVAR_SESSION_NULL, 580, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678 MPI_T_SCOPE_GROUP, 571, 678 MPI_T_SCOPE_GROUP, 571, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_STATE, 577, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_SESSION_NULL, 581, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678 MPI_T_SCOPE_CONSTANT, 571, 678 MPI_T_SCOPE_GROUP, 571, 678 MPI_T_SCOPE_GROUP_EQ, 571, 574, 678 MPI_T_SCOPE_LOCAL, 571, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_SESSION_NULL, 581, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678 MPI_T_SCOPE_GROUP, 571, 678 MPI_T_SCOPE_GROUP_EQ, 571, 574, 678 MPI_T_SCOPE_LOCAL, 571, 678 MPI_T_SCOPE_LOCAL, 571, 678
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_CLASS_TIMER, 577, 678 MPI_T_PVAR_SESSION_NULL, 580, 677 MPI_T_SCOPE_ALL, 571, 678 MPI_T_SCOPE_ALL_EQ, 571, 574, 678 MPI_T_SCOPE_CONSTANT, 571, 678 MPI_T_SCOPE_GROUP, 571, 678 MPI_T_SCOPE_GROUP_EQ, 571, 574, 678 MPI_T_SCOPE_LOCAL, 571, 678 MPI_T_SCOPE_READONLY, 571, 678 MPI_T_VERBOSITY_MPIDEV_ALL, 564,
<ul> <li>MPI_T_PVAR_CLASS_SIZE, 576, 678</li> <li>MPI_T_PVAR_CLASS_STATE, 575, 678</li> <li>MPI_T_PVAR_CLASS_TIMER, 577, 678</li> <li>MPI_T_PVAR_HANDLE_NULL, 581, 677</li> <li>MPI_T_SCOPE_ALL, 571, 678</li> <li>MPI_T_SCOPE_ALL_EQ, 571, 574, 678</li> <li>MPI_T_SCOPE_GROUP, 571, 678</li> <li>MPI_T_SCOPE_GROUP_EQ, 571, 574, 678</li> <li>MPI_T_SCOPE_LOCAL, 571, 678</li> <li>MPI_T_SCOPE_READONLY, 571, 678</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, 564, 677</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, 564,</li> </ul>
$\begin{array}{l} {\rm MPI\_T\_PVAR\_CLASS\_SIZE, 576, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_STATE, 575, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_BSSSION\_NULL, 581, 677} \\ {\rm MPI\_T\_SCOPE\_ALL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_ALL\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP, 571, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_LOCAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 677} \\ \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ \\ \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ \\ \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ \\ \\ \\ \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETA$
$\begin{array}{l} {\rm MPI\_T\_PVAR\_CLASS\_SIZE, 576, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_STATE, 575, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_SESSION\_NULL, 581, 677} \\ {\rm MPI\_T\_SCOPE\_ALL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_ALL\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP, 571, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_LOCAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ \\ \end{array}$
$\begin{array}{l} {\rm MPI\_T\_PVAR\_CLASS\_SIZE, 576, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_STATE, 575, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_SESSION\_NULL, 581, 677} \\ {\rm MPI\_T\_SCOPE\_ALL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_ALL\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP, 571, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNET\_AUL\_VERBOSITY\_TUNET$
$\begin{array}{l} {\rm MPI\_T\_PVAR\_CLASS\_SIZE, 576, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_STATE, 575, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_SESSION\_NULL, 581, 677} \\ {\rm MPI\_T\_SCOPE\_ALL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_ALL\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_LOCAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ASIC, 564, 674} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ASIC, 564, 674} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ASIC, 564, 674} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ASIC, 564, 676} \\ {\rm MPI\_T\_VERBOSITY\_TUN$
$\begin{array}{l} {\rm MPI\_T\_PVAR\_CLASS\_SIZE, 576, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_STATE, 575, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_SESSION\_NULL, 581, 677} \\ {\rm MPI\_T\_SCOPE\_ALL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_ALL\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_LOCAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_BASIC, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_BASIC, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_DETAIL, 564, 677} \\ {\rm MP$
$\begin{array}{l} {\rm MPI\_T\_PVAR\_CLASS\_SIZE, 576, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_STATE, 575, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_CLASS\_TIMER, 577, 678} \\ {\rm MPI\_T\_PVAR\_SESSION\_NULL, 581, 677} \\ {\rm MPI\_T\_SCOPE\_ALL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_ALL\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_CONSTANT, 571, 678} \\ {\rm MPI\_T\_SCOPE\_GROUP\_EQ, 571, 574, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_COAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_LOCAL, 571, 678} \\ {\rm MPI\_T\_SCOPE\_READONLY, 571, 678} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_MPIDEV\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_ALL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_BASIC, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_DETAIL, 564, 677} \\ {\rm MPI\_T\_VERBOSITY\_TUNER\_D$
$\begin{array}{llllllllllllllllllllllllllllllllllll$
$\begin{array}{llllllllllllllllllllllllllllllllllll$
$\begin{array}{llllllllllllllllllllllllllllllllllll$

MPI_T_VERBOSITY_USER_DETAIL, 564,	MPI_WIN_SIZE, 413, 414, 674	1
677	MPI_WIN_UNIFIED, 414, 436, 454, 463, 674	2
MPI_TAG, 30, 197, 668, 793	MPI_WTIME_IS_GLOBAL, 335, 336, 354,	3
MPI_TAG_UB, 27, 335, 657, 660, 671	657,671	4
MPI_THREAD_FUNNELED, 485, 486, 675		5
MPI_THREAD_MULTIPLE, 486, 488, 675		6
MPI_THREAD_SERIALIZED, 486, 675		
MPI_THREAD_SINGLE, 485–487, 675		7
MPI_TYPE_DUP_FN, <u>276</u> , 673		8
MPI_TYPE_NULL_COPY_FN, <u>276</u> , 673		9
MPI_TYPE_NULL_DELETE_FN, <u>276</u> , 673,		10
794		11
MPI_TYPECLASS_COMPLEX, 626, 676		12
MPI_TYPECLASS_INTEGER, 626, 676		13
MPI_TYPECLASS_REAL, 626, 676		14
MPI_UB, 4, 18, 600, 790		
MPI_UINT16_T, 26, 176, 669, 795–797		15
MPI_UINT32_T, 26, 176, 669, 795–797		16
MPI_UINT64_T, 26, 176, 669, 795–797		17
MPI_UINT8_T, 26, 176, 669, 795–797		18
MPI_UNDEFINED, 31, 58, 61, 62, 103, 106,		19
108, 113, 134, 228, 229, 244, 245, 302,		20
312, 313, 621, 622, 667, 791, 798		21
MPI_UNEQUAL, 229, 237, 259, 671		
MPI_UNIVERSE_SIZE, 374, 395, 396, 674		22
MPI_UNSIGNED, 26, 176, 567, 576, 577, 669		23
MPI_UNSIGNED_CHAR, 26, 176, 178, 179,		24
669		25
MPI_UNSIGNED_LONG, 26, 176, 567, 576,		26
577,669		27
MPI_UNSIGNED_LONG_LONG, 26, 176,		28
567, 576, 577, 669, 797		
MPI_UNSIGNED_SHORT, 26, 176, 669		29
MPI_UNWEIGHTED, 16, 297, 298, 300, 301,		30
308,  309,  628,  676,  790,  796		31
MPI_VAL, 13, 650		32
MPI_VERSION, 15, 334, 677		33
MPI_WCHAR, 26, 178, 179, 284, 534, 669, 797		34
MPI_WEIGHTS_EMPTY, 297, 298, 300, 628,		35
676, 790		36
MPI_WIN_BASE, 413, 414, 661, 674		37
MPI_WIN_CREATE_FLAVOR, 413, 414, 674		
MPI_WIN_DISP_UNIT, 413, 414, 674		38
MPI_WIN_DUP_FN, <u>273</u> , 673		39
MPI_WIN_FLAVOR_ALLOCATE, 414, 674		40
MPI_WIN_FLAVOR_CREATE, 414, 674		41
MPI_WIN_FLAVOR_DYNAMIC, 414, 674		42
MPI_WIN_FLAVOR_SHARED, 414, 674		43
MPI_WIN_MODEL, 413, 414, 436, 674		44
MPI_WIN_NULL, 413, 672		
MPI_WIN_NULL_COPY_FN, <u>273</u> , 673		45
MPI_WIN_NULL_DELETE_FN, <u>273</u> , 673		46
MPI_WIN_SEPARATE, 414, 435, 436, 454,		47
674		48

# **MPI Declarations Index**

 This index refers to declarations needed in C[ticket281.][/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).

13	MPI_Aint, <u>16</u> , 16, 17, 19, 25, <u>85</u> , 85, 88, 90, 93,
14	102, 106, 107, 117, 138, 139, 403, 405,
15	407, 410, 411, 418, 420, 424, 426,
16	428-433, 534, 537, 629, 657, 679
17	MPI_Comm, 13, <u>24</u> , 230, 235–240, <u>242</u> , 244,
18	$247, 248, \underline{249}, 259-262, 268, 270-272,$
19	671, 672, 679 MPI_Count, <u>17</u> , 17, 25, 679, 791
20	MPI_Datatype, <u>85</u> , 637, 669–672, 679
21	MPI_ERR, 347
21	MPI_Errhandler, <u>341</u> , 342–346, 651, 668, 672,
	679
23	MPI_F08_status, <u>653</u> , 677, 679, 793
24	MPI_File, 345, 346, 353, <u>491</u> , 493, 495–497,
25	499,501,504,507- $523,525$ - $530,534,$
26	542,543,651,672,679
27	MPI_Fint, <u>650</u> , 650, 677, 679, 797
28	MPI_Group, <u>228</u> , 228–235, 240, 260, 415, 441,
29	442, 497, 650, 651, 672, 679
30	MPI_Info, 337, <u>365</u> , 365–369, 374, 377, 380,
31	386, 388-392, 415, 416, 491, 494, 499,
32	501, 651, 671, 672, 679, 800 MPI_Message, <u>68</u> , 651, 667, 672, 679, 791
33	MPI_Offset, <u>17</u> , 17, 19, 25, 495, 496, 501, 504,
34	507-511, 516, 517, 522, 523, 525, 526,
35	538, 545, 545, 649, 679
36	MPI_Op, 174, <u>183</u> , 185, 187, 189–191, 193,
37	194, 209-214, 424, 426, 428, 432, 433,
38	651,672,679
39	MPI_Request, 49–51, <u>53</u> , 54, 55, 57–61, 64, 71,
40	74-77, 474, 477, 510, 511, 515, 516,
	519, 520, 631, 651, 672, 679
41	MPI_Status, <u>28</u> , 30–32, 53, 54, 57–61, 64, 65,
42	68-70, 72, 79, 80, 112, 113, 474, 480,
43	481, 507–509, 512–514, 518, 519, 521,
44	522, 525-530, 605, 652-654, 676, 679, 791, 793
45	MPI_T_cvar_handle, <u>572</u> , 572–574, 677
46	MPI_T_enum, <u>567</u> , 567, 568, 570, 578, 677
47	MPI_T_ERR, 590
48	

 $\begin{array}{l} \mathrm{MPI\_T\_pvar\_handle, } \underline{580}, 580-583, 677 \\ \mathrm{MPI\_T\_pvar\_session, } \underline{579}, 579-583, 677 \\ \mathrm{MPI\_Win, } 273-275, 285, 343, 344, 353, \underline{403}, \\ \underline{405}, \underline{407}, \underline{410}, 413, 415, 416, 418, 420, \\ 424, 426, 428-433, 439, 441-443, 445, \\ 446, 448, 449, 651, 672, 679 \\ \end{array}$ 

# 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. [ticket281.][C++ names for these typedefs and ]Fortran example prototypes are given near the text of the C name.

MPI_Comm_copy_attr_function, 18, 19, 268,	15
604,673,680	16
MPI_Comm_delete_attr_function, 18, 268,	17
673,680	18
MPI_Comm_errhandler_fn, 597, 796	19
MPI_Comm_errhandler_function, 18, 342, 597,	20
600,  680,  796	
MPI_Copy_function, 18, 594, 674, 685	21
MPI_Datarep_conversion_function, 538, 673,	22
680	23
MPI_Datarep_extent_function, 537, 680	24
MPI_Delete_function, 18, 595, 674, 685	25
MPI_File_errhandler_fn, 597, 796	26
MPI_File_errhandler_function, 345, 597, 680,	27
796	
MPI_Grequest_cancel_function, 476, 680	28
MPI_Grequest_free_function, 475, 680	29
MPI_Grequest_query_function, 475, 680	30
MPI_Handler_function, 18, 600, 790	31
MPI_Type_copy_attr_function, 276, 673, 680	32
MPI_Type_delete_attr_function, 276, 673,	33
680, 794	34
MPI_User_function, 183, 187, 680	35
MPI_Win_copy_attr_function, 273, 673, 680	
MPI_Win_delete_attr_function, 273, 673, 680	36
MPI_Win_errhandler_fn, 597, 796	37
MPI_Win_errhandler_function, 343, 597, 680,	38
796	39
	40

#### **MPI** Function Index

78 9 The underlined page numbers refer to the function definitions. 10 11MPI\_ABORT, 184, 340, 357, 360, 398, 567, 12650, 800 13MPI\_ACCUMULATE, 401, 416, <u>424</u>, 425, 427, 433, 436, 460, 466, 468, 797, 800 14MPI\_ADD\_ERROR\_CLASS, 350, 351 15MPI\_ADD\_ERROR\_CODE, 351 16MPI\_ADD\_ERROR\_STRING, 352, 352 17MPI\_ADDRESS, 18, 599, 614, 790 18MPI\_ALLGATHER, 141, 145, 146, 165, 19165-168, 204 MPI\_ALLGATHERV, 141, 145, 146, 166, 167, 2020521MPI\_ALLOC\_MEM, <u>337</u>, 338, 339, 348, 22404-409, 412, 419, 447, 611, 612, 23614-616, 629, 794  $^{24}$ MPI\_ALLOC\_MEM\_CPTR, 338, 611 25MPI\_ALLREDUCE, 141, 144–146, 176, 183, 26187, 188, 210, 798 MPI\_ALLTOALL, 141, 145, 146, <u>168</u>, 168-171, 27206, 795 28MPI\_ALLTOALLV, 141, 145, 146, 170, 170, 29 171, 173, 207, 795 30 MPI\_ALLTOALLW, 141, 145, 146, 172, 173,  $^{31}$ 209, 795 32 MPI\_ATTR\_DELETE, 18, 279, 595, 596 33 MPI\_ATTR\_GET, 18, 279, 596, 657, 658 34MPI\_ATTR\_PUT, 18, 279, 596, 657, 658, 660, 35661 MPI\_BARRIER, 141, 145, 147, 147, 148, 198, 36 457-459, 547 37 MPI\_BCAST, 141, 145, 148, 148, 149, 175, 38 199, 218 39 MPI\_BSEND, 38, 46 40MPI\_BSEND\_INIT, 74, 77  $^{41}$ MPI\_BUFFER\_ATTACH, 44, 53 42MPI\_BUFFER\_DETACH, 45, 793 MPI\_CANCEL, 41, 53, 64, 71, 71-73, 197, 358, 43430, 473, 476, 477  $^{44}$ MPI\_CART\_COORDS, 291, 305, 305, 799 45MPI\_CART\_CREATE, 258, 290, 291, 292, 46292-294, 304, 312-314, 630, 798 47MPI\_CART\_GET, 291, 304, 304, 799

5 6

48

MPI\_CART\_MAP, 291, 312, 313, 792 MPI\_CART\_RANK, 291, 305, 305, 799 MPI\_CART\_SHIFT, 291, 310, 310, 311, 314, 799 MPI\_CART\_SUB, 291, 311, 312, 313, 799 MPI\_CARTDIM\_GET, 291, 304, 304, 799 MPI\_CLOSE\_PORT, 387, 387, 391 MPI\_COMM\_ACCEPT, 386, 388, 388, 389, 396, 397 MPI\_COMM\_C2F, 650 MPI\_COMM\_CALL\_ERRHANDLER, 352, 354MPI\_COMM\_COMPARE, 237, 259 MPI\_COMM\_CONNECT, 348, 389, 389, 396, 397 MPI\_COMM\_CREATE, 235, 237, 240, 240-245, 291, 796 MPI\_COMM\_CREATE\_ERRHANDLER, 18, <u>341</u>, 341, 342, 599, 682, 684, 794 MPI\_COMM\_CREATE\_GROUP, 237, 242, 243, 244, 792 MPI\_COMM\_CREATE\_KEYVAL, 18, 266, 267, 269, 270, 279, 594, 656, 657, 681, 683, 794, 798 MPI\_COMM\_DELETE\_ATTR, 18, 266, 269-271, <u>272</u>, 279, 596 MPI\_COMM\_DISCONNECT, 279, 379, 397, <u>398</u>, 398 MPI\_COMM\_DUP, 230, 235, 237, 238, 238, 239, 241, 248, 249, 260, 262, 266, 269, 272, 279, 286, 594, 792 MPI\_COMM\_DUP\_FN, 18, 269, 269, 270, 612, 673, 794 MPI\_COMM\_DUP\_WITH\_INFO, 237, 238, 239, 248, 791 MPI\_COMM\_F2C, 650 MPI\_COMM\_FREE, 235, 238, 248, 248, 260, 262, 269, 270, 272, 279, 357, 360, 379, 397, 398, 595 MPI\_COMM\_FREE\_KEYVAL, 18, 266, 270, 279.595

- MPI\_COMM\_GET\_ATTR, 18, 266, 271, 271, 279, 334, 596, 657, 658, 660 MPI\_COMM\_GET\_ERRHANDLER, 18, 341, 343, 599, 799 MPI\_COMM\_GET\_INFO, 249, 250, 792 MPI\_COMM\_GET\_NAME, <u>282</u>, 282, 283, 798 MPI\_COMM\_GET\_PARENT, 283, 375, 378, 378, 379 MPI\_COMM\_GROUP, 14, 227, 230, 230, 235, 236, 259, 341, 800 MPI\_COMM\_IDUP, 235, 237, 239, 239, 248, 249, 257, 266, 269, 272, 279, 792 MPI\_COMM\_JOIN, 399, 399, 400 MPI\_COMM\_KEYVAL\_CREATE, 612 MPI\_COMM\_NULL\_COPY\_FN, 18, 269, 269, 270, 604, 656, 673, 794 MPI\_COMM\_NULL\_DELETE\_FN, 18, 269, 269, 270, 673 MPI\_COMM\_RANK, <u>236</u>, 236, 259, 612 MPI\_COMM\_REMOTE\_GROUP, 260 MPI\_COMM\_REMOTE\_SIZE, 260, 260 MPI\_COMM\_SET\_ATTR, 18, 266, 269, 270, 279, 595, 657, 658, 661 MPI\_COMM\_SET\_ERRHANDLER, 18, 341, <u>342</u>, 599 MPI\_COMM\_SET\_INFO, 248, 249, 249, 791 MPI\_COMM\_SET\_NAME, 281, 282 MPI\_COMM\_SIZE, 235, 236, 259 MPI\_COMM\_SPAWN, 356, 362, 363, 372, 373, 374, 374, 375, 377-379, 381-383, 395 - 397MPI\_COMM\_SPAWN\_MULTIPLE, 356, 362, 372, 373, 378, <u>380</u>, 381, 396, 397 MPI\_COMM\_SPLIT, 237, 240, 241, 244, 244-246, 286, 291, 292, 294, 312-314, 796 MPI\_COMM\_SPLIT\_TYPE, 247, 248, 792 MPI\_COMM\_TEST\_INTER, 258, 259 MPI\_COMM\_WORLD, 487 MPI\_COMPARE\_AND\_SWAP, 401, 417, 429, 466MPI\_CONVERSION\_FN\_NULL, 539, 673 MPI\_DIMS\_CREATE, 291, 293, 293 MPI\_DIST\_GRAPH\_CREATE, 248, 290, 291, 296, 298, 299, 301, 309, 310, 314, 796 MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, 248, 290, 291, 296, 296, 297, 301, 309, 314, 792, 796 MPI\_DIST\_GRAPH\_NEIGHBOR\_COUNT, 310 MPI\_DIST\_GRAPH\_NEIGHBORS, 291, 308, <u>309</u>, 309, 314, 792, 796
- MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT, 291, <u>308</u>, 308, 309, 790, 796

MPI_DUP_FN, 18, 269, <u>595</u> , 674	1
MPI_ERRHANDLER_C2F, $651$	2
MPI_ERRHANDLER_CREATE, 18, 599, 790,	3
794	4
MPI_ERRHANDLER_F2C, 651	
MPI_ERRHANDLER_FREE, 341, 346, 357,	5
799	6
MPI_ERRHANDLER_GET, 18, 599, 790, 800	7
MPI_ERRHANDLER_SET, 18, 599, 790	8
MPI_ERROR_CLASS, <u>347</u> , 347, 350, 590	9
MPI_ERROR_STRING, <u>346</u> , 347, 350, 350 MPI_ERROR_STRING, <u>346</u> , 347, 350, 352	
	10
MPI_EXSCAN, 142, 145, 176, 183, <u>194</u> , 194,	11
214, 796	12
MPI_F_SYNC_REG, 102, 602, 618, <u>619</u> , 620,	13
639-643,  645,  794	14
MPI_FETCH_AND_OP, 401, 416, 425, 427,	
428	15
$MPI\_FILE\_C2F, \underline{651}$	16
MPI_FILE_CALL_ERRHANDLER, <u>353</u> , 354	17
MPI_FILE_CLOSE, 398, 491, 492, 493, 494	18
MPI_FILE_CREATE_ERRHANDLER, 341,	19
<u>345</u> , 346, 682, 684, 794	
MPI_FILE_DELETE, 493, 494, 494, 498, 501,	20
550	21
MPI_FILE_F2C, <u>651</u>	22
MPI_FILE_GET_AMODE, <u>497</u> , 497	23
MPI_FILE_GET_AMODE, <u>497</u> MPI_FILE_GET_ATOMICITY, <u>543</u> , 543	24
MPI_FILE_GET_ATOMICTI 1, <u>545</u> , 545 MPI_FILE_GET_BYTE_OFFSET, 511, <u>517</u> ,	
	25
517, 518, 523	26
MPI_FILE_GET_ERRHANDLER, 341, <u>346</u> ,	27
550, 799	28
MPI_FILE_GET_GROUP, <u>497</u> , 497	29
MPI_FILE_GET_INFO, <u>499</u> , 499, 501, 801	30
MPI_FILE_GET_POSITION, $517$ , $517$	
MPI_FILE_GET_POSITION_SHARED, 522,	31
523, 523, 543	32
MPI_FILE_GET_SIZE, <u>496</u> , 497, 546	33
MPI_FILE_GET_TYPE_EXTENT, 533, 534,	34
540	35
MPI_FILE_GET_VIEW, <u>504</u> , 504	
MPI_FILE_IXXX, 506	36
MPI_FILE_IREAD, 505, <u>515</u> , 515, 523, 541	37
MPI_FILE_IREAD_AT, 505, 510, 510	38
MPI_FILE_IREAD_SHARED, 505, 519, 520	39
MPI_FILE_IWRITE, 505, <u>516</u> , 516	40
MPI_FILE_IWRITE_AT, 505, 511, 511	41
MPI_FILE_IWRITE_SHARED, 505, 520, 520	
MPI_FILE_OPEN, 349, 484, 491, 491–493,	42
$\begin{array}{c} \text{MF1\_FILE\_OFEN, 349, 404, \underline{491}, 491 \\ 498, 500 \\ -502, 518, 545, 550, 551 \end{array}$	43
	44
MPI_FILE_PREALLOCATE, 495, <u>496</u> , 496,	45
541, 545, 546	46
MPI_FILE_READ, 504, 505, <u>512</u> , 512, 513,	
515, 545, 546	47

1	MPI_FILE_READ_ALL, 505, 513, 513, 524,
2	525
3	MPI_FILE_READ_ALL_BEGIN, 505, 524,
4	525, 527, 541, 645
5	$MPI\_FILE\_READ\_ALL\_END, 505, 524, 525,$
6	527, 541, 645
7	MPI_FILE_READ_AT, 505, 507, 508, 510
8	MPI_FILE_READ_AT_ALL, 505, 508, 508
9	MPI_FILE_READ_AT_ALL_BEGIN, 505,
10	<u>525</u> , 645 MPI_FILE_READ_AT_ALL_END, 505, <u>525</u> ,
10	$\begin{array}{c} \text{MF1_FILE_READ_A1\_ALL\_END, 505, } \underline{525}, \\ 645 \end{array}$
	MPI_FILE_READ_ORDERED, 505, 521, 521
12	MPI_FILE_READ_ORDERED_BEGIN, 505,
13	<u>529</u> , 645
14	MPI_FILE_READ_ORDERED_END, 505,
15	529, 645
16	MPI_FILE_READ_SHARED, 505, <u>518</u> ,
17	519 - 521
18	MPI_FILE_SEEK, <u>516</u> , 517
19	MPI_FILE_SEEK_SHARED, <u>522</u> , 522, 523,
20	543 MPI_FILE_SET_ATOMICITY, 493, <u>542</u> , 542
21	MPI_FILE_SET_ATOMICTI 1, 493, <u>542</u> , 542 MPI_FILE_SET_ERRHANDLER, 341, 345,
22	550
23	MPI_FILE_SET_INFO, 498, <u>499</u> , 499–501, 800
24	MPI_FILE_SET_SIZE, <u>495</u> , 495, 496, 541,
25	544 - 546
26	MPI_FILE_SET_VIEW, 96, 349, 492, 498,
27	$500, \underline{501}, 501-503, 517, 523, 531, 537,$
28	545, 551, 800, 801
29	MPI_FILE_SYNC, 494, 505, 541, 542, <u>543</u> ,
30	543, 548 MPI_FILE_WRITE, 504, 505, <u>513</u> , 514, 516,
31	$\begin{array}{c} \text{MF1_FILE_WKITE, 504, 505, } \underline{515, 514, 510,} \\ 545 \end{array}$
32	MPI_FILE_WRITE_ALL, 505, <u>514</u> , 514
33	MPI_FILE_WRITE_ALL_BEGIN, 505, 528,
34	630, 645
35	MPI_FILE_WRITE_ALL_END, 505, <u>528</u> , 645
36	MPI_FILE_WRITE_AT, 505, <u>509</u> , 509–511
37	MPI_FILE_WRITE_AT_ALL, 505, <u>509</u> , 510
	MPI_FILE_WRITE_AT_ALL_BEGIN, 505,
38	<u>526, 609, 645</u>
39	MPI_FILE_WRITE_AT_ALL_END, 505, <u>526</u> ,
40	645 MPI_FILE_WRITE_ORDERED, 505, 521,
41	522, 522
42	MPI_FILE_WRITE_ORDERED_BEGIN,
43	505, 529, 645
44	MPI_FILE_WRITE_ORDERED_END, 505,
45	
	530, 645
46	MPI_FILE_WRITE_SHARED, 505, <u>519</u> ,
$\frac{46}{47}$	,

MPI_FINALIZE, 15, 22, 334, 335, <u>357</u> ,
357-361, 398, 483, 492, 563, 572, 584,
586, 650, 652, 653, 792, 800
MPI_FINALIZED, 355, 359, <u>361</u> , 361, 488, 650
MPI_FREE_MEM, <u>338</u> , 338, 348, 406, 407,
612
MPI_GATHER, 141, 144–146, <u>149</u> , 151, 152,
159,160,165,175,201
MPI_GATHERV, 141, 145, 146, <u>151</u> , 151–153,
161, 167, 202
MPI_GET, 401, 416, <u>420</u> , 421, 427, 432, 436,
457, 459, 460, 642, 800
MPI_GET_ACCUMULATE, 401, 416, 425,
$\underline{426}, 427, 434, 460, 466$
MPI_GET_ADDRESS, 18, 85, 101, <u>102</u> , 102,
114, 411, 599, 612, 614, 633, 637, 638,
655, 656
MPI_GET_COUNT, <u>31</u> , 31, 32, 52, 113, 114,
430, 481, 507, 791, 798
MPI_GET_ELEMENTS, 52, <u>112</u> , 112–114,
481, 482, 507, 791
MPI_GET_ELEMENTS_X, 52, 112, <u>113</u> , 113,
114, 481, 507, 791
MPI_GET_LIBRARY_VERSION, <u>334</u> , 334,
355, 359, 790
MPI_GET_PROCESSOR_NAME, <u>336</u> , <u>336</u> ,
337, 799
MPI_GET_VERSION, <u>333</u> , 334, 355, 359, 488,
617
MPI_GRAPH_CREATE, 290, 291, <u>294</u> , 294,
296, 300, 303, 306, 307, 313, 314, 798,
799
MPI_GRAPH_GET, 291, <u>303</u> , 303
MPI_GRAPH_MAP, 291, <u>313</u> , 314
MPI_GRAPH_NEIGHBORS, 291, <u>306</u> , 306,
307, 314, 796
MPI_GRAPH_NEIGHBORS_COUNT, 291,
<u>306</u> , 306, 307, 796
MPI_GRAPHDIMS_GET, 291, <u>303</u> , 303
MPI_GREQUEST_COMPLETE, 474–476,
477, 477
MPI_GREQUEST_START, <u>474</u> , 474, 682, 684,
797
MPI_GROUP_C2F, <u>651</u>
MPI_GROUP_COMPARE, 229, 232
MPI_GROUP_DIFFERENCE, 231
MPI_GROUP_EXCL, <u>232</u> , 233, 234
MPI_GROUP_F2C, $\underline{651}$
MPI_GROUP_FREE, <u>235</u> , 235, 236, 341, 357,
800
MPI_GROUP_INCL, <u>232</u> , 232, 234
MPI_GROUP_INTERSECTION, 231
MPI_GROUP_INTERSECTION, <u>231</u> MPI_GROUP_RANGE_EXCL, <u>234</u> , 234

MPI\_GROUP\_RANK, <u>228</u>, 236 MPI\_GROUP\_SIZE, <u>228</u>, 236 MPI\_GROUP\_TRANSLATE\_RANKS, 229, 229, 798 MPI\_GROUP\_UNION, 230 MPI\_IALLGATHER, 141, 145, 146, 204 MPI\_IALLGATHERV, 141, 145, 146, 205 MPI\_IALLREDUCE, 141, 145, 146, <u>210</u> MPI\_IALLTOALL, 141, 145, 146, 206 MPI\_IALLTOALLV, 141, 145, 146, 207 MPI\_IALLTOALLW, 141, 145, 146, 208 MPI\_IBARRIER, 141, 145, 197, <u>198</u>, 198, 199, 218 MPI\_IBCAST, 141, 145, <u>199</u>, 199, 222 MPI\_IBSEND, <u>49</u>, 53, 77 MPI\_IEXSCAN, 142, 145, 214 MPI\_IGATHER, 141, 145, 146, 200 MPI\_IGATHERV, 141, 145, 146, 201 MPI\_IMPROBE, 64, 67, 68, 68, 69, 71, 484, 791 MPI\_IMRECV, 67-69, 71, 71, 791 MPI\_INEIGHBOR\_ALLGATHER, 292, 324, 792 MPI\_INEIGHBOR\_ALLGATHERV, 292, 325, 792 MPI\_INEIGHBOR\_ALLTOALL, 292, 326, 792 MPI\_INEIGHBOR\_ALLTOALLV, 292, 327, 792 MPI\_INEIGHBOR\_ALLTOALLW, 292, 328, 792 MPI\_INFO\_C2F,  $\underline{651}$ MPI\_INFO\_CREATE, 366, 366 MPI\_INFO\_DELETE, 348, 367, 367, 369 MPI\_INFO\_DUP, 369, 369 MPI\_INFO\_F2C, 651 MPI\_INFO\_FREE, 250, 357, <u>369</u>, 416, 499 MPI\_INFO\_GET, 365, 367, 800 MPI\_INFO\_GET\_NKEYS, 365, 368, 368, 369, 800 MPI\_INFO\_GET\_NTHKEY, 365, 369, 800 MPI\_INFO\_GET\_VALUELEN, 365, 368, 800 MPI\_INFO\_SET, <u>366</u>, 366, 367, 369 MPI\_INIT, 15, 22, 227, 334, 335, 355, 355, 359, 361, 375–377, 379, 395, 396, 485, 486, 488, 559, 563, 566, 567, 572, 584, 590, 649, 650, 652, 653, 791, 792, 795, 797 MPI\_INIT\_THREAD, 227, 355, 361, 485, 486-488, 566, 590, 649, 791, 792, 797 MPI\_INITIALIZED, 355, <u>359</u>, 359, 361, 488, 650 MPI\_INTERCOMM\_CREATE, 237, 243, 260, 261, 261, 262, 792 MPI\_INTERCOMM\_MERGE, 237, 243, 258, 260, 261, 262, 262, 794

MPI_IPROBE, 31, <u>64</u> , 64–69, 71, 484, 791	1
MPI_IRECV, <u>51</u> , 71, 632, 635, 636	2
MPI_IREDUCE, 141, 145, 146, <u>209</u> , 209	3
MPI_IREDUCE_SCATTER, 141, 145, 146,	
212	4
MPI_IREDUCE_SCATTER_BLOCK, 141,	5
145, 146, <u>211</u>	6
MPI_IRSEND, <u>51</u>	7
MPI_IS_THREAD_MAIN, 485, 487	8
MPI_ISCAN, 142, 145, <u>213</u>	9
MPI_ISCAN, 142, 145, <u>215</u> MPI_ISCATTER, 141, 145, 146, <u>202</u>	
	10
MPI_ISCATTERV, 141, 145, 146, <u>203</u>	11
MPI_ISEND, 11, <u>49</u> , 77, 612, 613, 630, 632,	12
637 NDL LOCEND 50	13
MPI_ISSEND, <u>50</u>	14
MPI_KEYVAL_CREATE, 18, <u>594</u> , 596, 685	15
MPI_KEYVAL_FREE, 18, 279, <u>595</u>	
MPI_LOCK_ALL, 404	16
MPI_LOOKUP_NAME, 348, 386, 390, <u>392</u> ,	17
392	18
MPI_MESSAGE_C2F, <u>651</u> , 791	19
MPI_MESSAGE_F2C, <u>651</u> , 791	20
MPI_MPROBE, 64, 67, 68, <u>69</u> , 69, 71, 484, 791	21
MPI_MRECV, 67–69, <u>70</u> , 70, 71, 791	
MPI_NEIGHBOR_ALLGATHER, 291, <u>315</u> ,	22
317, 318, 324, 792	23
MPI_NEIGHBOR_ALLGATHERV, 291, <u>317</u> ,	24
325,792	25
MPI_NEIGHBOR_ALLTOALL, 291, <u>319</u> , 320,	26
326,792	27
MPI_NEIGHBOR_ALLTOALLV, 292, <u>320</u> ,	28
327,792	
MPI_NEIGHBOR_ALLTOALLW, 292, 321,	29
322, 329, 792	30
MPI_NULL_COPY_FN, 18, 19, 269, <u>595</u> , 674	31
MPI_NULL_DELETE_FN, 18, 269, <u>595</u> , 674	32
MPI_OP_C2F, <u>651</u>	33
MPI_OP_COMMUTATIVE, <u>189</u> , 796	34
MPI_OP_CREATE, <u>183</u> , <u>183</u> , <u>185</u> , <u>612</u> , <u>680</u> ,	
683, 794	35
MPI_OP_F2C, <u>651</u>	36
MPI_OP_FREE, <u>185</u> , 357	37
MPI_OPEN_PORT, <u>386</u> , 386, 388, 389, 391,	38
392	39
MPI_PACK, 46, <u>131</u> , 134, 137, 535, 538	40
MPI_PACK_EXTERNAL, 7, 137, <u>138</u> , 624,	
798	41
MPI_PACK_EXTERNAL_SIZE, <u>139</u>	42
MPI_PACK_EXTERNAL_SIZE, <u>139</u> MPI_PACK_SIZE, <u>46</u> , <u>134</u> , 134, 791	43
	44
MPI_PCONTROL, 558, <u>559</u> , 559	45
MPI_PROBE, 29, 31, 32, 64, <u>65</u> , 65–67, 69, 71,	46
484, 791 MDL DUDLISH NAME 286, 200, 200, 202	
MPI_PUBLISH_NAME, 386, <u>390</u> , 390–392	47
	48

1	MPI_PUT, 401, 416, <u>418</u> , 420, 424, 425, 431,
2	436, 442, 451, 453, 458, 459, 630, 642,
3	800
4	MPI_QUERY_THREAD, <u>487</u> , 488
5	MPI_RACCUMULATE, 401, 416, 417, 425,
	427, 432, 433
6	MPI_RECV, 24, 28, 30–32, 65, 67, 68, 84, 111,
7	112, 132, 142, 150, 219, 482, 547, 584,
8	637, 638, 641, 642
9	MPI_RECV_INIT, 76, 77
10	MPI_REDUCE, 141, 145, 146, <u>174</u> , 174–176,
11	183-185, 188, 191-194, 209, 425, 427,
	428, 797
12	MPI_REDUCE_LOCAL, 175, 176, 183, 189,
13	794, 796
14	MPI_REDUCE_SCATTER, 141, 145, 146,
15	176, 183, <u>191</u> , 191, 192, 212
16	MPI_REDUCE_SCATTER_BLOCK, 141,
17	145, 146, 176, 183, <u>190</u> , 190, 191, 211,
18	$\frac{110}{100}, \frac{110}{100}, \frac{100}{100}, \frac{100}{100}, \frac{100}{100}, \frac{100}{100}, \frac{100}{200}, \frac{100}{100}, 1$
	MPI_REGISTER_DATAREP, 349, 535,
19	537-539, 551, 612, 683, 685
20	MPI_REQUEST_C2F, <u>651</u>
21	$MPI_REQUEST_F2C, \underline{651}$
22	MPI_REQUEST_FREE, <u>55</u> , 55, 72, 78, 197,
23	357, 430, 476, 477, 795
24	MPI_REQUEST_GET_STATUS, 32, <u>64</u> , 64,
25	475, 795
26	MPI_RGET, 401, 416, 417, <u>431</u> , 432
27	MPI_RGET_ACCUMULATE, 401, 416, 417,
28	$425, 427, \underline{433}, 434$
	MPI_RPUT, 401, 416, 417, <u>430</u> , 431
29	MPI_RSEND, $39$
30	MPI_RSEND_INIT, <u>76</u>
31	MPI_SCAN, 142, 145, 176, 183, <u>193</u> , 193, 195,
32	213
33	MPI_SCATTER, 141, 145, 146, <u>159</u> , 159, 161,
34	162, 191, 202
35	MPI_SCATTERV, 141, 145, 146, <u>161</u> , 161,
36	162, 192, 203
37	MPI_SEND, 23, <u>24</u> , 25, 32, 34, 84, 111, 131,
38	219, 492, 547, 560, 609–613, 637, 638,
	641, 642
39	MPI_SEND_INIT, <u>74</u> , 77
40	MPI_SENDRECV, <u>79</u> , 310
41	MPI_SENDRECV_REPLACE, <u>80</u>
42	MPI_SIZEOF, 602, <u>625</u> , 626
43	MPI_SSEND, <u>39</u> MDI_SSEND_INIT_75
44	MPI_SSEND_INIT, <u>75</u> MPI_STADT_77_77_78
45	MPI_START, <u>77</u> , 77, 78
46	MPI_STARTALL, <u>77</u> , 78
	MPI_STATUS_C2F, <u>652</u> MDI_STATUS_C2F08_652_702
47	MPI_STATUS_C2F08, <u>653</u> , 793
48	MPI_STATUS_F082C, <u>653</u> , 793

MPI_STATUS_F082F, 654, 793
MPI_STATUS_F2C, 652
MPI_STATUS_F2F08, <u>654</u> , 793
MPI_STATUS_SET_CANCELLED, 481
MPI_STATUS_SET_ELEMENTS, <u>480</u> , 481
MPI_STATUS_SET_ELEMENTS_X, <u>481</u> ,
481, 791
MPI_T_CATEGORY_CHANGED, <u>589</u>
MPI_T_CATEGORY_GET_CATEGORIES,
<u>589</u> , 589–591
MPI_T_CATEGORY_GET_CVARS, <u>588</u> , 588,
590, 591
MPI_T_CATEGORY_GET_INFO, <u>588</u> , 590,
591
MPI_T_CATEGORY_GET_NUM, <u>587</u>
MPI_T_CATEGORY_GET_PVARS, <u>589</u> ,
589 - 591
MPI_T_CVAR_GET_INFO, 567, 569, 570,
570, 571, 573, 574, 590, 591
MPI_T_CVAR_GET_NUM, <u>569</u> , 573
MPI_T_CVAR_HANDLE_ALLOC, <u>572</u> , 573,
574, 591
MPI_T_CVAR_HANDLE_FREE, <u>573</u> , 573,
591
MPI_T_CVAR_READ, <u>573</u>
MPI_T_CVAR_WRITE, <u>574</u>
MPI_T_ENUM_GET_INFO, <u>568</u> , 568, 591
MPI_T_ENUM_GET_ITEM, <u>568</u> , 568, 591
MPI_T_FINALIZE, <u>566</u> , 566, 567
MPI_T_INIT_THREAD, <u>565</u> , 566, 567
MPI_T_PVAR_GET_INFO, 567, 577, <u>578</u> ,
578, 580 - 584, 590, 591
MPI_T_PVAR_GET_NUM, <u>578</u> , 581
MPI_T_PVAR_HANDLE_ALLOC, <u>580</u> , <u>580</u> ,
582, 583, 591
MPI_T_PVAR_HANDLE_FREE, <u>581</u> , 581,
591
MPI_T_PVAR_READ, <u>582</u> , 582, 583, 591
MPI_T_PVAR_READRESET, 579, <u>583</u> , 583,
591
MPI_T_PVAR_RESET, <u>583</u> , 583, 591
$\begin{array}{c} \text{MIII_I_I} \text{ VAR}_\text{RESEI}, \underline{303}, \underline{303}, \underline{391} \\ \text{MDIT DVAD CECCION ODEATE 570 501} \end{array}$
MPI_T_PVAR_SESSION_CREATE, <u>579</u> , 591
MPI_T_PVAR_SESSION_FREE, <u>580</u> , 591
MPI_T_PVAR_START, <u>581</u> , 591
MPI_T_PVAR_STOP, <u>582</u> , 591
MPI_T_PVAR_WRITE, <u>582</u> , 583, 591
MPI_TEST, 11, 32, 52, 53, <u>54</u> , 54, 55, 57, 58,
72, 78, 357, 477, 506, 507
MPI_TEST_CANCELLED, 52–54, <u>72</u> , 73, 475,
482, 507
MPI_TESTALL, 57, <u>60</u> , 60, 475–477, 480, 483
MPI_TESTANY, 53, 57, <u>58</u> , 58, 62, 475–477,
480, 483
,

MPI\_TESTSOME, 57, <u>61</u>, 62, 475–477, 480, 483 MPI\_TOPO\_TEST, 291, 302, 302 MPI\_TYPE\_C2F, 650 MPI\_TYPE\_COMMIT, 109, 109, 651 MPI\_TYPE\_CONTIGUOUS, 12, 85, 85, 87, 104, 116, 490, 533 MPI\_TYPE\_CREATE\_DARRAY, 12, 31, 97, 97, 98, 116 MPI\_TYPE\_CREATE\_F90\_COMPLEX, 12, 116, 118, 177, 535, 602, 622, 623, 624 MPI\_TYPE\_CREATE\_F90\_INTEGER, 12, 116, 118, 176, 535, 602, 622, 623, 624 MPI\_TYPE\_CREATE\_F90\_REAL, 12, 116, 118, 177, 535, 602, <u>621</u>, 621–624, 795 MPI\_TYPE\_CREATE\_HINDEXED, 12, 18, 85, 90, 90, 92, 94, 116, 599 MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK,  $12, 85, \underline{92}, 92, 116, 791$ MPI\_TYPE\_CREATE\_HVECTOR, 12, 18, 85, 88, 88, 116, 599 MPI\_TYPE\_CREATE\_INDEXED\_BLOCK, 12, 91, 92, 116 MPI\_TYPE\_CREATE\_KEYVAL, 266, 276, 279, 657, 681, 684, 798 MPI\_TYPE\_CREATE\_RESIZED, 18, 85, 104, 107, 107, 116, 533, 600, 793 MPI\_TYPE\_CREATE\_STRUCT, 12, 18, 85, 92, 93, 93, 94, 104, 116, 173, 599 MPI\_TYPE\_CREATE\_SUBARRAY, 12, 15, 94, 96, 98, 116 MPI\_TYPE\_DELETE\_ATTR, 266, 279, 279, 794 MPI\_TYPE\_DUP, 12, <u>110</u>, 110, 116, 794 MPI\_TYPE\_DUP\_FN, 276, 276, 673 MPI\_TYPE\_EXTENT, 18, 599, 790 MPI\_TYPE\_F2C, <u>650</u> MPI\_TYPE\_FREE, 110, 118, 277, 357 MPI\_TYPE\_FREE\_KEYVAL, 266, 277, 279 MPI\_TYPE\_GET\_ATTR, 266, 278, 279, 657, 794 MPI\_TYPE\_GET\_CONTENTS, 116, 117, 117 - 119MPI\_TYPE\_GET\_ENVELOPE, 115, 116, 118, 119, 623 MPI\_TYPE\_GET\_EXTENT, 18, 106, 108, 599, 626, 655 MPI\_TYPE\_GET\_EXTENT\_X, 106, 791 MPI\_TYPE\_GET\_NAME, <u>284</u>, 794 MPI\_TYPE\_GET\_TRUE\_EXTENT, 107, 107 MPI\_TYPE\_GET\_TRUE\_EXTENT\_X, 107, 108, 791 MPI\_TYPE\_HINDEXED, 18, 599, 790 MPI\_TYPE\_HVECTOR, 18, 599, 790

MPI_TYPE_INDEXED, 12, <u>89</u> , 89–91, 116	1
MPI_TYPE_LB, 18, 599, 790	2
MPI_TYPE_MATCH_SIZE, 602, <u>625</u> , 626, 794	3
MPI_TYPE_NULL_COPY_FN, <u>276</u> , 276, 673	4
MPI_TYPE_NULL_DELETE_FN, <u>276</u> , 673,	
794	5
MPI_TYPE_SET_ATTR, 266, <u>278</u> , 279, 657,	6
661, 794	7
MPI_TYPE_SET_NAME, <u>284</u> , 794	8
MPI_TYPE_SIZE, <u>103</u> , 103, 560, 791	9
MPI_TYPE_SIZE_X, <u>103</u> , 103, 791	10
MPI_TYPE_STRUCT, 18, 599, 790	11
MPI_TYPE_UB, 18, 599, 790	
MPI_TYPE_VECTOR, 12, <u>86</u> , 86–88, 90, 116	12
MPI_UNPACK, <u>132</u> , 132, 133, 137, 538	13
MPI_UNPACK_EXTERNAL, 7, <u>138</u> , 624	14
MPI_UNPUBLISH_NAME, 348, <u>391</u> , 391	15
MPI_WAIT, 30, 32, 52, <u>53</u> , 53–56, 58, 59, 72,	16
	17
78, 197, 219, 357, 473, 477, 484, 506,	
507, 523, 541, 542, 630, 631, 636, 637,	18
641 MDI WAITALL 57 50 50 60 107 990 420	19
MPI_WAITALL, 57, <u>59</u> , 59, 60, 197, 220, 430,	20
475-477, 480, 483, 484	21
MPI_WAITANY, 41, 53, <u>57</u> , 57, 58, 62,	22
475-477, 480, 483, 484	23
MPI_WAITSOME, 57, <u>60</u> , 61–63, 475–477,	
480, 483, 484	24
MPI_WIN_ALLOC, 408	25
MPI_WIN_ALLOCATE, 402, <u>405</u> , 406, 413,	26
414, 419, 447, 612, 615, 616	27
MPI_WIN_ALLOCATE_CPTR, 406	28
MPI_WIN_ALLOCATE_SHARED, 402, <u>407</u> ,	29
407, 409, 413, 414, 616	30
MPI_WIN_ALLOCATE_SHARED_CPTR,	
408	31
MPI_WIN_ATTACH, 410, <u>411</u> , 411, 412, 447	32
$MPI_WIN_C2F, \underline{651}$	33
MPI_WIN_CALL_ERRHANDLER, <u>353</u> , 354	34
MPI_WIN_COMPLETE, 413, 437, <u>441</u> ,	35
442-444, 452, 459	36
MPI_WIN_CREATE, 402, <u>403</u> , 405, 406, 408,	37
410, 412-414, 451, 484	
MPI_WIN_CREATE_DYNAMIC, 349, 402,	38
$\underline{410},  410 - 414,  452$	39
MPI_WIN_CREATE_ERRHANDLER, 341,	40
$\underline{343}, 344, 682, 684, 794$	41
MPI_WIN_CREATE_KEYVAL, 266, <u>272</u> , 279,	42
657,681,683,798	43
MPI_WIN_DELETE_ATTR, 266, <u>275</u> , 279	
MPI_WIN_DETACH, 410, <u>412</u> , 412, 413	44
MPI_WIN_DUP_FN, <u>273</u> , 273, 673	45
MPI_WIN_F2C, <u>651</u>	46
MPI_WIN_FENCE, 413, 421, 436, <u>439</u> ,	47
439-441, 450-453, 456, 461, 642	48

1	MPI_WIN_FLUSH, 408, 430, 431, <u>448</u> , 448,
2	452, 455, 466, 468
3	MPI_WIN_FLUSH_ALL, 430, 431, <u>448</u> , 452,
4	455
5	MPI_WIN_FLUSH_LOCAL, 430, <u>449</u> , 452
6	MPI_WIN_FLUSH_LOCAL_ALL, 430, <u>449</u> ,
7	449, 452 MPI_WIN_FREE, 274, 357, 398, <u>413</u> , 413
8	MPI_WIN_FREE, 274, 357, 398, <u>415</u> , 415 MPI_WIN_FREE_KEYVAL, 266, <u>274</u> , 279
9	MPI_WIN_GET_ATTR, 266, <u>275</u> , 279, 414,
10	657, 661
11	MPI_WIN_GET_ERRHANDLER, 341, 344,
12	799
13	MPI_WIN_GET_GROUP, <u>415</u> , 415
14	MPI_WIN_GET_INFO, <u>416</u> , 416, 792
15	MPI_WIN_GET_NAME, <u>285</u> MPI_WIN_LOCK, 404, 437, <u>445</u> , 446–448,
16	$\begin{array}{c} \text{MF1}_{\text{WIN}} \text{IOCK}, \ 404, \ 457, \ \underline{440}, \ 440-448, \\ 450, \ 451, \ 453, \ 456-458 \end{array}$
17	MPI_WIN_LOCK_ALL, 437, <u>445</u> , 446, 450,
18	451, 453, 458, 466, 467
19	MPI_WIN_NULL_COPY_FN, <u>273</u> , 273, 673
20	MPI_WIN_NULL_DELETE_FN, <u>273</u> , 673
21	MPI_WIN_POST, 413, 437, 441, <u>442</u> , 442–445,
22	447, 450, 453, 459, 461 MPI_WIN_SET_ATTR, 266, 274, 279, 414,
23	$\begin{array}{c} \text{MF1}_{\text{WIN}_{\text{SE1}_{\text{A}11\text{R}}}, 200, \underline{274}, 279, 414, \\ 657, 661 \end{array}$
24	MPI_WIN_SET_ERRHANDLER, 341, 344
25	MPI_WIN_SET_INFO, <u>415</u> , 415, 416, 792
26	MPI_WIN_SET_NAME, <u>285</u>
27	MPI_WIN_SHARED_ALLOCATE, 612
28	MPI_WIN_SHARED_QUERY, 407, <u>409</u> , 612,
29	616 MPI_WIN_SHARED_QUERY_CPTR, 410
30	MPI_WIN_START, 437, <u>441</u> , 441, 442, 444,
31	445, 450, 459, 466
32	MPI_WIN_SYNC, <u>449</u> , 449, 453, 454, 459,
33	466, 468
34	MPI_WIN_TEST, <u>443</u> , 444
35	MPI_WIN_UNLOCK, 431, 437, <u>446</u> , 447, 448,
36	452, 453, 456, 457 MPI_WIN_UNLOCK_ALL, 431, 437, <u>446</u> ,
37	$\begin{array}{c} \text{MP1}_{\text{WIN}_{\text{ONLOCK}_{\text{ALL}}}, 451, 451, 457, \underline{440}, \\ 452, 453, 456, 467 \end{array}$
38	MPI_WIN_WAIT, 413, 437, <u>443</u> , 443, 444,
39	447, 452, 453, 456, 459, 460
40	MPI_WTICK, 20, <u>355</u> , 355
41	MPI_WTIME, 20, 336, <u>354</u> , 354, 355, 560, 577
42	mpiexec, 356, 360, <u>362</u> , 362, 486
43	mpirun, 361, 362
44	PMPI_, 557
45	PMPI_SEND, 611
46	PMPI_WTICK, 20
47	PMPI_WTIME, 20
48	