## MPI: A Message-Passing Interface Standard Version 3.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

July 27, 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 mpi-comments@mpi-forum.org. Your com-
16	ment will be forwarded to MPI Forum committee members who will attempt to respond.
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	©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.

Version 3.0: xx, x, 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 to make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes from Version 1.0 are minor. A version of this document with all changes marked is available.

Unofficial Draft for Comment Only

 $\mathbf{5}$ 

 $^{24}$ 

Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation  $^{2}$ from over 40 organizations, has been meeting since January 1993 to discuss and to define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.  $\mathbf{5}$ 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  $\overline{7}$ practical, portable, efficient, and flexible standard for message-passing. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LATEX on May 5, 1994. 

# Contents

A	cknov	vledgments	$\mathbf{i}\mathbf{x}$
1	Intr	oduction to MPI	1
	1.1	Overview and Goals	1
	1.2	Background of MPI-1.0	2
	1.3	Background of MPI-1.1, MPI-1.2, and MPI-2.0	2
	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?	5
	1.9	What Is Included In The Standard?	5
	1.10	What Is Not Included In The Standard?	6
		Organization of this Document	6
2		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	14
		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	16
	2.6	Language Binding	17
		2.6.1 Deprecated and Removed Names and Functions	17
		2.6.2 Fortran Binding Issues	18
		2.6.3 C Binding Issues	19
		2.6.4 Functions and Macros	19
	2.7	Processes	20
	2.8	Error Handling	$\frac{1}{20}$
	2.9	Implementation Issues	21

	2.9.1	Independence of Basic Runtime Routines .							•			•		21
	2.9.2	Interaction with Signals												22
2.10	Examp													22
р.,														0.0
														23
														23
3.2														24
														24
														25
														27
														28
														30
														32
3.3	Data 7	· ·												33
	3.3.1	Type Matching Rules	•	•			•		•	•		•		33
		$Type MPI\_CHARACTER \dots \dots \dots \dots \dots$		•			•		•					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														44
	3.6.1	<u> </u>												46
3.7	Nonblo													47
														48
		* * *												48
														52
														56
														57
														63
20														64
3.0														64
														67
		<u> </u>												
														69
2.0														71
														73 70
														78
3.11	Null P	rocesses	•	·	•	•••	·	• •	•	•	• •	•	•	81
Dat	atypes													83
4.1	Derive	d Datatypes							•					83
	4.1.1	Type Constructors with Explicit Addresses												85
	4.1.2													85
	4.1.3													95
														97
														102
														102
														104
		· · · · · · · · · · · · · · · · · · ·												107
			•	•	•	•••	·	• •	•	•	• •	·	·	110
	Poir 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.7 3.8 3.8 3.9 3.10 3.11 Dat	2.9.2 2.10 Examp Point-to-P 3.1 Introd 3.2 Blockii 3.2.1 3.2.2 3.2.3 3.2.4 3.2.5 3.2.6 3.3 Data T 3.3.1 3.3.2 3.4 Comm 3.5 Seman 3.6 Buffer 3.6.1 3.7 Nonble 3.7.1 3.7.2 3.7.3 3.7.4 3.7.5 3.7.6 3.8 Probe 3.8.1 3.8.2 3.8.3 3.8.4 3.9 Persist 3.10 Send-H 3.11 Null P Datatypes 4.1 Derive 4.1.1 4.1.2 4.1.3 4.1.4 4.1.5 4.1.6 4.1.7 4.1.8	2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Data         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type Matching Rules         Type MPL_CHARACTER       3.3.2         3.3.1       Type Matching Rules         Type MPL_CHARACTER       3.3.2         3.3.2       Data Conversion         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6.1       Model Implementation of Buffered Mode         3.7       Nonblocking Communication         3.7.1       Communication Request Objects         3.7.2       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status <tr< td=""><td>2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Envelope         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPL_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type Matching Rules         Type MPL_CHARACTER       3.3.2         Data Conversion       3.3.1         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6       Buffer Allocation and Usage         3.7.1       Communication Initiation         3.7.2       Communication Request Objects         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8.1       Probe</td><td>2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Envelope         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type MPI_CHARACTER         3.3.2       Data Conversion         3.3.1       Type MPI_CHARACTER         3.3.2       Data Conversion         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6.1       Model Implementation of Buffered Mode         3.7.1       Communication Request Objects         3.7.2       Communication Completion         3.7.3       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8       Probe         3.8</td><td>2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Envelope         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching Rules         Type MPI_CHARACTER       3.3.2         Jota Conversion       3.3.1         Type MPI_CHARACTER       3.3.2         Jota Conversion       3.4         Communication Modes       3.5         Semantics of Point-to-Point Communication       3.6.1         Model Implementation of Buffered Mode       3.7.1         Communication Request Objects       3.7.2         Communication Completion       3.7.4         Semantics of Nonblocking Communications       3.7.5         Multiple Completions       3.7.6         Non-destructive Test of status       3.8.1         Probe and Cancel       3.8.3         Matched Receives       3.8.4         Cancel</td><td>2.9.2 Interaction with Signals         2.10 Examples         2.11 Examples         2.11 Introduction         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Data         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Request Objects         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8.1 Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3.10 Send-Receive         3.11 Null Processes         3.10 Send-Receive         3.11 Null Processes         4.1.1 Type Cons</td><td>2.9.2       Interaction with Signals         2.10       Examples         2.11       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Data         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type Matching Rules         Type MPI_CHARACTER       3.3.2         Data Conversion       3.3.1         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6.1       Model Implementation of Buffered Mode         3.7       Nonblocking Communication         3.7.1       Communication Request Objects         3.7.2       Communication Completion         3.7.3       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8.1       Probe</td><td>2.9.2 Interaction with Signals         2.10 Examples         2.11 Examples         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Request Objects         3.7.3 Communication Completion         3.7.4 Semantics of Noblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8.1 Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.8.3 Matched Receives         3.8.4 Cancel         3.8.3 Matched Receives         3.8.4 Cancel         3.8.5 Probe         3.8.6 Probe         3.8.7 Desistent Communication Requests</td><td>2.9.2       Interaction with Signals         2.10       Examples         Point-to-Point Communication         3.1       Introduction         3.2       Blocking Send         3.2.1       Blocking Send         3.2.2       Message Data         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPL_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type MPL_CHARACTER         3.3.2       Data Conversion         3.3.1       Type MPL_CHARACTER         3.3.2       Data Conversion         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6       Buffer Allocation and Usage         3.6.1       Model Implementation of Buffered Mode         3.7.1       Communication Request Objects         3.7.2       Communication Request Objects         3.7.3       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8</td><td>2.9.2 Interaction with Signals         2.10 Examples         2.10 Examples         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPI_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPI_CHARACTER         3.3.2 Data Conversion         3.3.1 Type Matching Rules         Type MPI_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Initiation         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8 Probe and Cancel         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3</td><td>2.9.2 Interaction with Signals         2.10 Examples         2.11 Examples         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6 Buffer Allocation and Usage         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Request Objects         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8 Probe and Cancel         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.</td><td>2.9.2 Interaction with Signals         2.10 Examples         Point-to-Point Communication         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPI_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching and Data Conversion         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6 In Model Implementation of Buffered Mode         3.7.1 Communication Initiation         3.7.2 Communication Initiation         3.7.3 Communication Initiation         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8 Probe and Cancel         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3.10 Send-Receive         3.11 Null Processes         Datatypes         4.1 Derived Datatypes</td><td>2.9.2 Interaction with Signals         2.10 Examples         Point-to-Point Communication         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching and Data Conversion         3.3.2 Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6 I Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Initiation         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Nultiple Completions         3.7.6 Non-destructive Test of status         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3.10 Send-Receive         3.11 Null Processes</td></tr<>	2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Envelope         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPL_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type Matching Rules         Type MPL_CHARACTER       3.3.2         Data Conversion       3.3.1         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6       Buffer Allocation and Usage         3.7.1       Communication Initiation         3.7.2       Communication Request Objects         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8.1       Probe	2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Envelope         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type MPI_CHARACTER         3.3.2       Data Conversion         3.3.1       Type MPI_CHARACTER         3.3.2       Data Conversion         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6.1       Model Implementation of Buffered Mode         3.7.1       Communication Request Objects         3.7.2       Communication Completion         3.7.3       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8       Probe         3.8	2.9.2       Interaction with Signals         2.10       Examples         2.10       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Envelope         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching Rules         Type MPI_CHARACTER       3.3.2         Jota Conversion       3.3.1         Type MPI_CHARACTER       3.3.2         Jota Conversion       3.4         Communication Modes       3.5         Semantics of Point-to-Point Communication       3.6.1         Model Implementation of Buffered Mode       3.7.1         Communication Request Objects       3.7.2         Communication Completion       3.7.4         Semantics of Nonblocking Communications       3.7.5         Multiple Completions       3.7.6         Non-destructive Test of status       3.8.1         Probe and Cancel       3.8.3         Matched Receives       3.8.4         Cancel	2.9.2 Interaction with Signals         2.10 Examples         2.11 Examples         2.11 Introduction         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Data         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Request Objects         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8.1 Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3.10 Send-Receive         3.11 Null Processes         3.10 Send-Receive         3.11 Null Processes         4.1.1 Type Cons	2.9.2       Interaction with Signals         2.10       Examples         2.11       Examples         3.1       Introduction         3.2       Blocking Send and Receive Operations         3.2.1       Blocking Send         3.2.2       Message Data         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPI_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type Matching Rules         Type MPI_CHARACTER       3.3.2         Data Conversion       3.3.1         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6.1       Model Implementation of Buffered Mode         3.7       Nonblocking Communication         3.7.1       Communication Request Objects         3.7.2       Communication Completion         3.7.3       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8.1       Probe	2.9.2 Interaction with Signals         2.10 Examples         2.11 Examples         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Request Objects         3.7.3 Communication Completion         3.7.4 Semantics of Noblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8.1 Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.8.3 Matched Receives         3.8.4 Cancel         3.8.3 Matched Receives         3.8.4 Cancel         3.8.5 Probe         3.8.6 Probe         3.8.7 Desistent Communication Requests	2.9.2       Interaction with Signals         2.10       Examples         Point-to-Point Communication         3.1       Introduction         3.2       Blocking Send         3.2.1       Blocking Send         3.2.2       Message Data         3.2.3       Message Envelope         3.2.4       Blocking Receive         3.2.5       Return Status         3.2.6       Passing MPL_STATUS_IGNORE for Status         3.3       Data Type Matching and Data Conversion         3.3.1       Type MPL_CHARACTER         3.3.2       Data Conversion         3.3.1       Type MPL_CHARACTER         3.3.2       Data Conversion         3.4       Communication Modes         3.5       Semantics of Point-to-Point Communication         3.6       Buffer Allocation and Usage         3.6.1       Model Implementation of Buffered Mode         3.7.1       Communication Request Objects         3.7.2       Communication Request Objects         3.7.3       Communication Completion         3.7.4       Semantics of Nonblocking Communications         3.7.5       Multiple Completions         3.7.6       Non-destructive Test of status         3.8	2.9.2 Interaction with Signals         2.10 Examples         2.10 Examples         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPI_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPI_CHARACTER         3.3.2 Data Conversion         3.3.1 Type Matching Rules         Type MPI_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Initiation         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8 Probe and Cancel         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3	2.9.2 Interaction with Signals         2.10 Examples         2.11 Examples         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6 Buffer Allocation and Usage         3.6.1 Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Request Objects         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8 Probe and Cancel         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.	2.9.2 Interaction with Signals         2.10 Examples         Point-to-Point Communication         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPI_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching and Data Conversion         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6 In Model Implementation of Buffered Mode         3.7.1 Communication Initiation         3.7.2 Communication Initiation         3.7.3 Communication Initiation         3.7.4 Semantics of Nonblocking Communications         3.7.5 Multiple Completions         3.7.6 Non-destructive Test of status         3.8 Probe and Cancel         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3.10 Send-Receive         3.11 Null Processes         Datatypes         4.1 Derived Datatypes	2.9.2 Interaction with Signals         2.10 Examples         Point-to-Point Communication         3.1 Introduction         3.2 Blocking Send and Receive Operations         3.2.1 Blocking Send         3.2.2 Message Data         3.2.3 Message Envelope         3.2.4 Blocking Receive         3.2.5 Return Status         3.2.6 Passing MPL_STATUS_IGNORE for Status         3.3 Data Type Matching and Data Conversion         3.3.1 Type Matching and Data Conversion         3.3.2 Data Conversion         3.3.1 Type Matching Rules         Type MPL_CHARACTER         3.3.2 Data Conversion         3.4 Communication Modes         3.5 Semantics of Point-to-Point Communication         3.6 I Model Implementation of Buffered Mode         3.7.1 Communication Request Objects         3.7.2 Communication Initiation         3.7.3 Communication Completion         3.7.4 Semantics of Nonblocking Communications         3.7.5 Nultiple Completions         3.7.6 Non-destructive Test of status         3.8.1 Probe         3.8.2 Matching Probe         3.8.3 Matched Receives         3.8.4 Cancel         3.9 Persistent Communication Requests         3.10 Send-Receive         3.11 Null Processes

		4.1.10	Duplicating a Datatype						111
		4.1.11	Use of General Datatypes in Communication						112
		4.1.12	Correct Use of Addresses						115
		4.1.13	Decoding a Datatype						116
			Examples						123
	4.2		nd Unpack						132
	4.3		ical MPI_PACK and MPI_UNPACK						138
5	Coll	lective	Communication						141
	5.1	Introd	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						145
		5.2.3	Specifics for Intercommunicator Collective Operations						146
	5.3	Barrie	r Synchronization						147
	5.4	Broade	cast						148
		5.4.1	Example using MPI_BCAST						149
	5.5	Gather	· · · · · · · · · · · · · · · · · · ·						149
		5.5.1	Examples using MPI_GATHER, MPI_GATHERV						152
	5.6	Scatter	r,						159
		5.6.1	Examples using MPI_SCATTER, MPI_SCATTERV						162
	5.7		r-to-all						165
		5.7.1	Example using MPI_ALLGATHER						167
	5.8		All Scatter/Gather						168
	5.9		Reduction Operations						173
	0.0	5.9.1	Reduce						174
		5.9.2	Predefined Reduction Operations						176
		5.9.3	Signed Characters and Reductions						178
		5.9.4	MINLOC and MAXLOC						179
		5.9.5	User-Defined Reduction Operations						183
		0.0.0	Example of User-defined Reduce						186
		5.9.6	All-Reduce						187
		5.9.7	Process-Local Reduction						189
	5 10		e-Scatter						190
	0.10		MPI_REDUCE_SCATTER_BLOCK						190
			MPI_REDUCE_SCATTER						190
	5 11								191
	0.11		Inclusive Scan						193
			Exclusive Scan						$193 \\ 194$
			Example using MPI_SCAN						$194 \\ 195$
	E 19								195 196
	0.12		Ocking Collective Operations						
			Nonblocking Barrier Synchronization						198
		0.12.2	Nonblocking Broadcast						199
		E 10.9	Example using MPI_IBCAST						200
			Nonblocking Gather						200
			Nonblocking Scatter						202
			Nonblocking Gather-to-all						204
		5.12.6	Nonblocking All-to-All Scatter/Gather	• •	•	•••	• •	·	206

		5.12.7	Nonblocking Reduce	09
		5.12.8	Nonblocking All-Reduce	10
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks	11
		5.12.10	Nonblocking Reduce-Scatter	12
		5.12.11	Nonblocking Inclusive Scan	13
				14
	5.13			14
	~			
6				23
	6.1			23
		6.1.1		23
	<b>C D</b>	6.1.2		24
	6.2		*	26
		6.2.1	1	26
		6.2.2		26
		6.2.3		27
		6.2.4		27
	6.3	_	8	28
		6.3.1	1	28
		6.3.2	*	30
		6.3.3	Group Destructors	35
	6.4	Comm	9	35
		6.4.1	Communicator Accessors	35
		6.4.2	Communicator Constructors	37
		6.4.3	Communicator Destructors	48
		6.4.4	Communicator Info	48
	6.5	Motiva	ting Examples	50
		6.5.1	Current Practice $\#1$	50
		6.5.2	Current Practice $\#2$	51
		6.5.3	(Approximate) Current Practice #3	51
		6.5.4		52
		6.5.5	Library Example #1	53
		6.5.6		55
	6.6	Inter-C		57
		6.6.1	Inter-communicator Accessors	59
		6.6.2	Inter-communicator Operations	60
		6.6.3	*	63
				63
				64
	6.7	Cachir		65
	0.1	6.7.1		66
		6.7.2	•	67
		6.7.3		72
		6.7.4		76
		6.7.4 6.7.5		79
		6.7.6	U U	79
	6 9		1	
	6.8 6.0			81 85
	6.9	rorma	lizing the Loosely Synchronous Model	00

		6.9.1	Basic Statements	285
		6.9.2	Models of Execution	286
			Static communicator allocation	286
			Dynamic communicator allocation	286
			The General Case	
7	Pro	cess 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	view 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	
		7.5.6	Cartesian Shift Coordinates	
		7.5.7	Partitioning of Cartesian Structures	
		7.5.8	Low-Level Topology Functions	312
	7.6		borhood Collective Communication on Process Topologies	314
	1.0	7.6.1	Neighborhood Gather	315
		7.6.2	Neighbor Alltoall	319
	7.7		locking Neighborhood Communication on Process Topologies	$313 \\ 324$
	1.1	7.7.1	Nonblocking Neighborhood Gather	
	7 0	7.7.2	Nonblocking Neighborhood Alltoall	
	7.8	All Aj	pplication Example	<b>3</b> 30
8	MPI	Envi	ronmental Management	<b>335</b>
	8.1		mentation Information	335
		8.1.1	Version Inquiries	
		8.1.2	Environmental Inquiries	
			Tag Values	
			Host Bank	337
			IO Rank	337
			Clock Synchronization	338
			Inquire Processor Name	338
	8.2	Mome	bry Allocation	339
	8.3		Handling	342
	0.0	8.3.1	Error Handlers for Communicators	343
		8.3.2	Error Handlers for Windows	$345 \\ 345$
		8.3.3	Error Handlers for Files	347
	0 1	8.3.4	Freeing Errorhandlers and Retrieving Error Strings	348
	8.4		Codes and Classes	349
	8.5		Classes, Error Codes, and Error Handlers	352
	8.6		rs and Synchronization	356
	8.7	Startu		357
		8.7.1	Allowing User Functions at Process Termination	363

		8.7.2	Determining Whether MPI Has Finished	63
	8.8	Portab	ble MPI Process Startup	64
9	The	Info C	Dbject 36	67
10	Pro	cess C	reation and Management 37	73
	10.1	Introd	$\operatorname{uction}$	73
	10.2	The D	ynamic Process Model	74
		10.2.1	Starting Processes	74
		10.2.2	The Runtime Environment	74
	10.3			76
		10.3.1	Processes in MPI	76
		10.3.2	Starting Processes and Establishing Communication	76
		10.3.3	Starting Multiple Executables and Establishing Communication 3	81
		10.3.4	Reserved Keys	84
		10.3.5	Spawn Example	85
			Manager-worker Example Using MPI_COMM_SPAWN	85
	10.4	Establ	ishing Communication	87
		10.4.1	Names, Addresses, Ports, and All That 34	87
		10.4.2	Server Routines	88
		10.4.3	Client Routines	90
		10.4.4	Name Publishing	92
		10.4.5	Reserved Key Values	94
		10.4.6	Client/Server Examples	94
			Simplest Example — Completely Portable	94
			Ocean/Atmosphere - Relies on Name Publishing 39	95
				95
	10.5	Other	Functionality	97
		10.5.1	Universe Size	97
		10.5.2	Singleton MPI_INIT 39	98
		10.5.3	MPI_APPNUM	98
		10.5.4	Releasing Connections	99
		10.5.5	Another Way to Establish MPI Communication	01
11				03
	11.1	Introd		03
	11.2	Initiali		04
				04
				07
				08
				11
				14
				15
				16
	11.3			18
		11.3.1		19
		11.3.2		21
		11.3.3	Examples for Communication Calls	22

	11.3.4 Accumulate Functions	4
	Accumulate Function 42	4
	Get Accumulate Function	7
	Fetch and Op Function	8
	Compare and Swap Function	9
	11.3.5 Request-based RMA Communication Operations	0
11.4	Memory Model	5
	Synchronization Calls	7
	11.5.1 Fence	0
	11.5.2 General Active Target Synchronization	2
	11.5.3 Lock	6
	11.5.4 Flush and Sync	9
	11.5.5 Assertions	
	11.5.6 Miscellaneous Clarifications	
11.6	Error Handling	
1110	11.6.1 Error Handlers	
	11.6.2 Error Classes	
11.7	Semantics and Correctness	
11.1	11.7.1 Atomicity	
	11.7.2 Ordering	
	11.7.3 Progress	
	11.7.4 Registers and Compiler Optimizations	
11.8	Examples	
11.0		9
<b>12</b> Exte	ernal Interfaces 47	5
12.1	Introduction	5
12.2	Generalized Requests	5
	12.2.1 Examples	0
12.3	Associating Information with Status	2
	MPI and Threads	4
	12.4.1 General	4
	12.4.2 Clarifications	
	12.4.3 Initialization	
		-
13 I/O	49	1
13.1	Introduction	1
	13.1.1 Definitions	1
13.2	File Manipulation	3
	13.2.1 Opening a File	3
	13.2.2 Closing a File	5
	13.2.3 Deleting a File	6
	13.2.4 Resizing a File	7
	13.2.5 Preallocating Space for a File	8
	13.2.6 Querying the Size of a File	8
	13.2.7 Querying File Parameters	
	13.2.8 File Info	
	Reserved File Hints	
13.3	File Views	
		- 1

13	.4 Data A	Access	 			 •		506
	13.4.1	Data Access Routines	 			 •		506
		Positioning	 					507
		Synchronism	 			 		508
		Coordination						508
		Data Access Conventions						508
	13.4.2	Data Access with Explicit Offsets						509
		Data Access with Individual File Pointers .						513
		Data Access with Shared File Pointers						520
	10.1.1	Noncollective Operations						521
		Collective Operations						523
		Seek						524
	1345	Split Collective Data Access Routines						526
13		teroperability						532
10		Datatypes for File Interoperability						534
		External Data Representation: "external32"						536
		User-Defined Data Representation. external 52						537
	10.0.0	Extent Callback						539
								539 540
	19 5 4	Datarep Conversion Functions						$540 \\ 542$
10		Matching Data Representations						-
13		tency and Semantics						542
		File Consistency						542
		Random Access vs. Sequential Files						545
		Progress						546
		Collective File Operations						546
	13.6.5	<i></i>						546
		Miscellaneous Clarifications						547
		MPI_Offset Type						547
		Logical vs. Physical File Layout						547
		File Size						547
	13.6.10	Examples						548
		Asynchronous I/O						550
		$rror Handling \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$						552
13	.8 I/O Ei	$\operatorname{rror} \operatorname{Classes} \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$	 	•••		 •		553
13		ples						553
	13.9.1	Double Buffering with Split Collective I/O	 			 •	•	553
	13.9.2	Subarray Filetype Constructor	 			 •		555
	ool Supp							557
		uction						557
14		ng Interface						557
		Requirements						557
		Discussion						558
		Logic of the Design						558
		Miscellaneous Control of Profiling						559
		Profiler Implementation Example						560
	14.2.6	MPI Library Implementation Example	 			 •		560
		Systems with Weak Symbols	 			 		560

	Systems Without Weak Symbols	561
	14.2.7 Complications	561
	Multiple Counting	561
	Linker Oddities	562
	Fortran Support Methods	562
	14.2.8 Multiple Levels of Interception	562
14.3	The MPI Tool Information Interface	563
	14.3.1 Verbosity Levels	564
	14.3.2 Binding MPI Tool Information Interface Variables to MPI Objects .	564
	14.3.3 Convention for Returning Strings	565
	14.3.4 Initialization and Finalization	565
	14.3.5 Datatype System	567
	14.3.6 Control Variables	569
	Control Variable Query Functions	569
	Example: Printing All Control Variables	571
	Handle Allocation and Deallocation	572
	Control Variable Access Functions	573
	Example: Reading the Value of a Control Variable	574
	14.3.7 Performance Variables	575
	Performance Variable Classes	575
	Performance Variable Query Functions	577
	Performance Experiment Sessions	579
	Handle Allocation and Deallocation	580
	Starting and Stopping of Performance Variables	581
	Performance Variable Access Functions	582
	Example: Tool to Detect Receives with Long Unexpected Message	
	Queues	584
	14.3.8 Variable Categorization	586
	14.3.9 Return Codes for the MPI tool information interface	590
	14.3.10 Profiling Interface	590
		593
	Deprecated since MPI-2.0	593
15.2	Deprecated since MPI-2.2	596
16 <b>Bon</b>	noved Interfaces	597
	Removed MPI-1 Bindings	597
10.1	16.1.1 Overview	597
	16.1.2 Removed MPI-1 Functions	597
	16.1.3 Removed MPI-1 Datatypes	597
	16.1.4 Removed MPI-1 Constants	598
	16.1.5 Removed MPI-1 Callback Prototyes	598
16.2	C++ Bindings	598
10.2		
17 Lan	guage Bindings	599
17.1	Fortran Support	599
	17.1.1 Overview	599
	17.1.2 Fortran Support Through the mpi_f08 Module	600

	17.1.3	Fortran Support Through the mpi Module	603
	17.1.4	Fortran Support Through the mpif.h Include File	605
	17.1.5	Interface Specifications, Linker Names and the Profiling Interface	607
	17.1.6	MPI for Different Fortran Standard Versions	611
	17.1.7	Requirements on Fortran Compilers	615
		Additional Support for Fortran Register-Memory-Synchronization	616
		Additional Support for Fortran Numeric Intrinsic Types	617
		Parameterized Datatypes with Specified Precision and Exponent Rang	
		Support for Size-specific MPI Datatypes	-
		Communication With Size-specific Types	624
	17 1 10	Problems With Fortran Bindings for MPI	625
		Problems Due to Strong Typing	627
		Problems Due to Data Copying and Sequence Association with Sub-	•
		script Triplets $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$	628
	17.1.15	Problems Due to Data Copying and Sequence Association with Vector	020
	1	Subscripts	630
	17 1 14	Special Constants	631
		5 Fortran Derived Types	631
		Optimization Problems, an Overview	633
		Problems with Code Movement and Register Optimization	634
		Nonblocking operations	634
		One-sided communication	635
		MPI_BOTTOM and combining independent variables in datatypes .	635
		Solutions	636
		The Fortran ASYNCHRONOUS attribute	637
		Calling MPI_F_SYNC_REG	639
		A user defined routine instead of MPI_F_SYNC_REG	640
		Module variables and COMMON blocks	640
		The (poorly performing) Fortran VOLATILE attribute	641
		The Fortran TARGET attribute	641
	17 1 18	Temporary Data Movement and Temporary Memory Modification	
		Permanent Data Movement	643
		Comparison with C	644
17.2		age Interoperability	647
11.2	<u> </u>	Introduction	647
		Assumptions	647
		Initialization	647
		Transfer of Handles	648
		Status	650
		MPI Opaque Objects	652
	11.2.0	Datatypes	653
		Callback Functions	654
		Error Handlers	655
		Reduce Operations	655
	17.2.7	Attributes	655
		Extra State	659
		Constants	659
		Interlanguage Communication	660
			000

Α	Lan	guage	Bindings Summary	663
	A.1	Define	d Values and Handles	. 663
		A.1.1	Defined Constants	. 663
		A.1.2		
		A.1.3	Prototype Definitions	
			C Bindings	
			Fortran 2008 Bindings with the mpi_f08 Module	
			Fortran Bindings with mpif.h or the mpi Module	
		A.1.4		
		A.1.5	Info Keys	
		A.1.6	Info Values	
	Δ 2		dings	
	<b>A.</b> 2		Point-to-Point Communication C Bindings	
		A.2.1 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		
			One-Sided Communications C Bindings	
			External Interfaces C Bindings	
			I/O C Bindings	
			2 Language Bindings C Bindings	
			B Tools / Profiling Interface C Bindings	
			Tools / MPI Tool Information Interface C Bindings	
			Deprecated C Bindings	
	A.3	Fortra	n 2008 Bindings with the mpi_f08 Module	. 709
		A.3.1	Point-to-Point Communication Fortran 2008 Bindings	. 709
		A.3.2	Datatypes Fortran 2008 Bindings	. 714
		A.3.3	Collective Communication Fortran 2008 Bindings	. 719
		A.3.4	Groups, Contexts, Communicators, and Caching Fortran 2008 Bindin	ngs726
		A.3.5	Process Topologies Fortran 2008 Bindings	. 733
		A.3.6	MPI Environmental Management Fortran 2008 Bindings	. 738
		A.3.7	The Info Object Fortran 2008 Bindings	. 741
		A.3.8	Process Creation and Management Fortran 2008 Bindings	. 742
		A.3.9	One-Sided Communications Fortran 2008 Bindings	. 743
		A.3.10	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	
		A.4.1	Point-to-Point Communication Fortran Bindings	
		A.4.2	Datatypes Fortran Bindings	
		A.4.3	Collective Communication Fortran Bindings	
		A.4.4	Groups, Contexts, Communicators, and Caching Fortran Bindings	
		A.4.5	Process Topologies Fortran Bindings	
		A.4.5 A.4.6	MPI Environmental Management Fortran Bindings	
		л.4.0	init i Duvnonmentai management rottian Dindings	. 114

A.4.7 The Info Object Fortran Bindings	6
A.4.8 Process Creation and Management Fortran Bindings	7
A.4.9 One-Sided Communications Fortran Bindings	8
A.4.10 External Interfaces Fortran Bindings	1
A.4.11 I/O Fortran Bindings	2
A.4.12 Language Bindings Fortran Bindings	6
A.4.13 Tools / Profiling Interface Fortran Bindings	7
A.4.14 Deprecated Fortran Bindings	7
	~
B Change-Log 789	-
B.1 Changes from Version 2.2 to Version 3.0	
B.1.1 Errata to Previous Versions of MPI	
B.1.2 Changes in MPI-3.0 $\dots$ 79	
B.2 Changes from Version 2.1 to Version 2.2	5
B.3 Changes from Version 2.0 to Version 2.1	7
Bibliography 80	3
Examples Index 80	8
MPI Constant and Predefined Handle Index 81	1
MPI Declarations Index 81	7
	0
MPI Callback Function Prototype Index 818	ð
MPI Function Index 81	9

# 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
	Overlapping Communicators Example	221
6.1	Intercommunicator create 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
$7.1 \\ 7.2$	Set-up of process structure for two-dimensional parallel Poisson solver Communication routine with local data copying and sparse neighborhood	331 332
7.9	all-to-all.	<u>3</u> 32
7.3	Communication routine with sparse neighborhood all-to-all-w and without	กกก
	local data copying.	333
11.1	Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows	436
11.2	Active target communication	439
	Active target communication, with weak synchronization	440
	Passive target communication	441
	Active target communication with several processes	444
	Symmetric communication	463
	Deadlock situation	463
	No deadlock	463
191	Etem og op di filotem og	492
	Etypes and filetypes	492 492
	Partitioning a file among parallel processes	
	Displacements	505
13.4	Example array file layout	555

13.5	Example local array filetype for process 1	•••	•	 •	•	•	•	• •	•	•	•	•	•	•	•	•	556
17.1	Status conversion routines			 •	•	•					•	•	•	•			651

# List of Tables

2.1	Deprecated and removed constructs	18
3.1 3.2 3.3 3.4	Predefined MPI datatypes corresponding to Fortran datatypes Predefined MPI datatypes corresponding to C datatypes Predefined MPI datatypes corresponding to both C and Fortran datatypes Predefined MPI datatypes corresponding to C++ datatypes	25 26 27 27
4.1	combiner values returned from MPI_TYPE_GET_ENVELOPE	118
6.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	259
8.1 8.2	Error classes (Part 1)Error classes (Part 2)	$\begin{array}{c} 350\\ 351 \end{array}$
	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR.	$\begin{array}{c} 415\\ 453 \end{array}$
13.2	Data access routines	$507 \\ 538 \\ 553$
$14.2 \\ 14.3 \\ 14.4$	MPI tool information interface verbosity levels	564 565 567 571 591
$\begin{array}{c} 16.2\\ 16.3 \end{array}$	Removed MPI-1 functions and their replacementsRemoved MPI-1 datatypes and their replacementsRemoved MPI-1 constantsRemoved MPI-1 callback prototypes and their replacements	597 598 598 598
17.1	Occurrence of Fortran optimization problems	634

## Acknowledgments

6

 $\overline{7}$ 

8

9 10 11

12

13

14

18 19

20 21

22 23

 $^{24}$ 

25 26

27

28 29

30 31

32 33

34

35

36 37

38 39

40

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.

Unofficial Draft for Comment Only

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz	1
Scott Berryman	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 James Kohl	C.T. Howard Ho Susan Krauss	Gary Howell Bob Leary	John Kapenga Arthur Maccabe	6 7
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley	8
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer	9
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison	10
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson	11
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith	12
				13
The Universit	v of Tennessee and (	)ak Ridge National 1	Laboratory made the draft avail-	14
	0	0	cal in distributing the document.	15
			t by ARPA and NSF under grant	16 17
ASC-9310330, the	National Science Fou	indation Science and	Technology Center Cooperative	17
-		the Commission of th	e European Community through	19
Esprit project P66	643 (PPPE).			20
				21
	<b>.</b>			22
MPI-1.2 and MPI-	-2.0:			23
Those who served	as primary coordina	tors in $MPI-1.2$ and	MPI-2.0 are:	24
	~			25
• Ewing Lusk,	Convener and Meet	ing Chair		26 27
• Steve Huss-L	ederman, Editor			28
	N.C. 11			29
• Ewing Lusk,	Miscellany			30
• Bill Saphir, l	Process Creation and	d Management		31
				32
• Marc Snir, C	Dne-Sided Communic	cations		33
• Bill Gropp a	nd Anthony Skjellur	n, Extended Collect	ive Operations	34
• Ctorro II.uga I	adamaan Ertanall	[mtonfo.cog		35 36
• Steve Huss-L	ederman, External l	Interfaces		37
• Bill Nitzberg	s, I/O			38
• Andrew Lum	nsdaine, Bill Saphir,	and leff Souvres L	anguaga Bindings	39
• Andrew Lun	istame, Dii Sapini,	and Jen Squyres, La	anguage Dinumgs	40
• Anthony Skj	ellum and Arkady K	anevsky, Real-Time		41
	1	C . 1		42
0			ants who attended MPI-2 Forum	43 44
meetings and ale i	not mentioned above	ē.		44 45
				46
				47
				48

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	0	1 1 1 1 1	• • • • • • • • •				
31		also acknowledges and ap	preciates the valuable	e input from people via			
32	e-mail and in person.						
33	The fall and in a			·····			
34	0	titutions supported the N	viPi-2 enort through	time and travel support			
35	for the people listed a	ibove.					
36	Argonne Nationa	l Laboratory					
37	Bolt, Beranek, ar	nd Newman					
38	California Institute of Technology						
39	Center for Computing Sciences						
40	Convex Computer Corporation						
41	Cray Research						
42	Digital Equipment Corporation						
43	Dolphin Interconnect Solutions, Inc.						
44	Edinburgh Parallel Computing Centre						
45	General Electric	* *					
46	German National Research Center for Information Technology						
47	Hewlett-Packard						
48	Hitachi						

1 2 3 4 5
3 4
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

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.

MPI-1.3 and MPI-2.1:
The editors and organizers of the combined documents have been:
• Richard Graham, Convener and Meeting Chair
• Jack Dongarra, Steering Committee
• Al Geist, Steering Committee
• Bill Gropp, Steering Committee
• Rainer Keller, Merge of MPI-1.3
• Andrew Lumsdaine, Steering Committee
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)
<ul> <li>Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)</li> </ul>
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:
• Bill Gropp, Frontmatter, Introduction, and Bibliography
• Richard Graham, Point-to-Point Communication
• Adam Moody, Collective Communication
• Richard Treumann, Groups, Contexts, and Communicators
• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions
• George Bosilca, Environmental Management
• David Solt, Process Creation and Management
• Bronis R. de Supinski, External Interfaces, and Profiling
• Rajeev Thakur, I/O
• Jeffrey M. Squyres, Language Bindings and MPI 2.1 Secretary
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log
• Alexander Supalov and Denis Nagorny, Annex Language Bindings
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.

Pavan Balaji Richard Barrett Gil Bloch Darius Buntinas Terry Dontje Karl Feind David Gingold Robert Harrison Torsten Hoefler Matthew Koop Miron Livny Avneesh Pant	Purushotham V. Bangalore Christian Bell Ron Brightwell Jonathan Carter Gabor Dozsa Edgar Gabriel Dave Goodell Thomas Herault Joshua Hursey Quincey Koziol Kannan Narasimhan Steve Poole	Brian Barrett Robert Blackmore Jeffrey Brown Nathan DeBardeleben Edric Ellis Patrick Geoffray Erez Haba Steve Hodson Yann Kalemkarian Sameer Kumar Mark Pagel Howard Pritchard	1 2 3 4 5 6 7 8 9 10 11 11
Craig Rasmussen Tony Skjellum	Hubert Ritzdorf Brian Smith	Rob Ross Vinod Tipparaju	13 14
Jesper Larsson Träff	Keith Underwood		15 16
The MPI Forum a e-mail and in person.	lso acknowledges and apprecia	tes the valuable input from people via	10 17 18
The following inst	itutions supported the MPI-2	effort through time and travel support	19
for the people listed at		shore sincuga since and staver support	20 21
Argonne National	Laboratory		21
Bull			23
Cisco Systems, In	с.		24
Cray Inc.			25
The HDF Group			26
Hewlett-Packard			27
IBM T.J. Watson			28 29
Indiana University Institut National	y de Recherche en Informatique	et Automatique (INRIA)	30
Intel Corporation		et Mutomatique (mutany)	31
-	y National Laboratory		32
	ore National Laboratory		33
Los Alamos Natio	nal Laboratory		34
Mathworks			35
Mellanox Technol	ogies		36
Microsoft			37 38
Myricom NEC Laboratorios	s Europe, NEC Europe Ltd.		39
Oak Ridge Nation			40
Ohio State Univer			41
	National Laboratory		42
QLogic Corporation			43
Sandia National L	Laboratories		44
SiCortex	· · · ·		45
Silicon Graphics I	-		46
Sun Microsystems University of Alak	s, Inc. Dama at Birmingham		47 48
University of Alac			

1	University of Houston
2	University of Illinois at Urbana-Champaign
3	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
4	University of Tennessee, Knoxville
5	University of Wisconsin
6	
7	Funding for the MPI Forum meetings was partially supported by award $\#$ CCF-0816909
8 9	from the National Science Foundation. In addition, the HDF Group provided travel support
10	for one U.S. academic.
11	
12	
13	MPI-2.2:
14 15 16	All chapters have been revisited to achieve a consistent MPI-2.2 text. Those who served as authors for the necessary modifications are:
17 18	• William Gropp, Frontmatter, Introduction, and Bibliography; MPI 2.2 chair.
19	• Richard Graham, Point-to-Point Communication and Datatypes
20 21	• Adam Moody, Collective Communication
22 23	• Torsten Hoefler, Collective Communication and Process Topologies
24	• Richard Treumann, Groups, Contexts, and Communicators
25 26	• Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications
27 28	• George Bosilca, Datatypes and Environmental Management
29 30	• David Solt, Process Creation and Management
31	• Bronis R. de Supinski, External Interfaces, and Profiling
32 33	• Rajeev Thakur, I/O
34	• Jeffrey M. Squyres, Language Bindings and MPI 2.2 Secretary
35 36 37	• Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language Bindings
38 39	• Alexander Supalov, Annex Language Bindings
40	The following list includes some of the active participants who attended MPI-2 Forum
41	meetings and in the e-mail discussions of the errata items and are not mentioned above.
42	
43	
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 MDI Ferrum also	admowledges and appreciates	the valuable input from people vie	22
	acknowledges and appreciates	s the valuable input from people via	23
e-mail and in person.			24
The following institut	tions supported the MPL 2.2 of	fort through time and travel support	25
_		fort through time and travel support	26
for the people listed abov Argonne National La			27
Auburn University	ibbratory		28
Bull			29
Cisco Systems, Inc.			30
Cray Inc.			31
Forschungszentrum J	liilich		32
Fujitsu	Junen		33
The HDF Group			34
Hewlett-Packard			35
International Busines	ss Machinos		36
Indiana University	55 Wachines		37
	Recherche en Informatique et	Automatique (INRIA)	38
	ed Science & Engineering Cor		39
Intel Corporation	ed belence & Engineering Corj	poration	40
Lawrence Berkeley N	lational Laboratory		41
	National Laboratory		42
Los Alamos National	Ũ		43
Mathworks	i Laboratory		44
Mellanox Technologi	65		45
Microsoft			46
Myricom			47
wryncom			48

1	NEC Corporation
2	Oak Ridge National Laboratory
3	Ohio State University
4	Pacific Northwest National Laboratory
5	QLogic Corporation
6	RunTime Computing Solutions, LLC
7	Sandia National Laboratories
8	SiCortex, Inc.
9	Silicon Graphics Inc.
10	Sun Microsystems, Inc.
11	
12	Tokyo Institute of Technology
	University of Alabama at Birmingham
13	University of Houston
14	University of Illinois at Urbana-Champaign
15	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
16	University of Tennessee, Knoxville
17	University of Tokyo
18	University of Wisconsin
19	Funding for the MPI Forum meetings was partially supported by award $\#$ CCF-0816909
20	
21	and $\#$ CCF-1144042 from the National Science Foundation. In addition, the HDF Group
22	provided travel support for one U.S. academic.
23	
24	
25	MPI-3:
26	MPI-3 is a significant effort to extend and modernize the MPI Standard.
27	The editors and organizers of the MPI-3 have been:
28	The editors and organizers of the Will-5 have been.
29	• William Gropp, Steering committee, Frontmatter, Introduction, Groups, Contexts,
30	and Communicators, One-Sided Communications, and Bibliography
31	
32	• Richard Graham, Steering committee, Point-to-Point Communication, Meeting Con-
33	vener, and MPI-3 chair
34	
35	• Torsten Hoefler, Collective Communication, One-Sided Communications, and Process
36	Topologies
37	• George Bosilca, Datatypes and Environmental Management
38	• George Dosnica, Datatypes and Environmental Management
39	• David Solt, Process Creation and Management
40	
	• Bronis R. de Supinski, External Interfaces and Tool Support
41	
42	• Rajeev Thakur, I/O and One-Sided Communications
43	• Darius Buntinas, Info Object
44	
45	• Jeffrey M. Squyres, Language Bindings and MPI 3.0 Secretary
46	
47 48	• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings

• Craig Rasmussen, Fortran I	Bindings		1
The following list includes so	mo of the active partici	ipants who attended MPI-3 Forum	2
meetings or participated in the e-	_	-	3
incettings of participated in the e-			4
Tatsuya Abe	Tomoya Adachi	Sadaf Alam	5
Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore	6
Brian Barrett	Richard Barrett	Robert Blackmore	7
Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky	8 9
Jed Brown	Darius Buntinas	Devendar Bureddy	
Arno Candel	George Carr	Mohamad Chaarawi	10 11
Raghunath Raja Chandrasekar	James Dinan	Terry Dontje	11
Edgar Gabriel	Balazs Gerofi	Brice Goglin	12
David Goodell	Manjunath Gorentla	Erez Haba	13
Jeff Hammond	Thomas Herault	Marc-André Hermanns	14
Jennifer Herrett-Skjellum	Nathan Hjelm	Atsushi Hori	16
Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa	10
Nysal Jan	Bin Jia	Hideyuki Jitsumoto	18
Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima	19
Chulho Kim	Dries Kimpe	Christof Klausecker	20
Alice Koniges	Quincey Koziol	Dieter Kranzlmueller	20
Manojkumar Krishnan	Sameer Kumar	Eric Lantz	21
Jay Lofstead	Bill Long	Andrew Lumsdaine	23
Miao Luo	Ewing Lusk	Adam Moody	24
Nick M. Maclaren	Amith Mamidala	Guillaume Mercier	25
Scott McMillan	Douglas Miller	Kathryn Mohror	26
Tim Murray	Tomotake Nakamura	Takeshi Nanri	27
Steve Oyanagi	Mark Pagel	Swann Perarnau	28
Sreeram Potluri	Howard Pritchard	Rolf Riesen	29
Hubert Ritzdorf Martin Schulz	Kuninobu Sasaki Gilad Shainer	Timo Schneider	30
		Christian Siebert Marc Snir	31
Anthony Skjellum Boffaele Ciuganna Salaa	Brian Smith		32
Raffaele Giuseppe Solca	Shinji Sumimoto Magamishi Takari	Alexander Supalov Fabian Tillier	33
Sayantan Sur Vined Tinparaiu	Masamichi Takagi Jesper Larsson Träff	Richard Treumann	34
Vinod Tipparaju Keith Underwood	Rolf Vandevaart	Anh Vo	35
Abhinav Vishnu	Min Xie		36
Abiinav visiniu	MIII Ale	Enqiang Zhou	37
The MPI Forum also acknow	ledges and appreciates t	the valuable input from people via	38
e-mail and in person.			39
			40
The following institutions su	pported the $MPI-3$ effor	t through time and travel support	41
for the people listed above.			42
Argonne National Laborator	У		43
Bull			44
Cisco Systems, Inc.			45
Cray Inc.			46
$\mathbf{CSCS}$			47
ETH Zurich			48

1	Fujitsu Ltd.
2	German Research School for Simulation Sciences
3	The HDF Group
4	Hewlett-Packard
5	International Business Machines
6	IBM India Private Ltd
7	Indiana University
8	Institut National de Recherche en Informatique et Automatique (INRIA)
9	Institute for Advanced Science & Engineering Corporation
10	Intel Corporation
11	Lawrence Berkeley National Laboratory
12	Lawrence Livermore National Laboratory
13	Los Alamos National Laboratory
14	Mellanox Technologies, Inc.
15	Microsoft Corporation
16	NEC Corporation
17	National Oceanic and Atmospheric Administration, Global Systems Division
18	NVIDIA Corporation
19	Oak Ridge National Laboratory
20	The Ohio State University
21	Oracle America
22	Platform Computing
23	RIKEN AICS
24	RunTime Computing Solutions, LLC
25	Sandia National Laboratories
26	Technical University of Chemnitz
27	Tokyo Institute of Technology
28	University of Alabama at Birmingham
29	University of Chicago
30	University of Houston
31	University of Illinois at Urbana-Champaign
32	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
33	University of Tennessee, Knoxville
34	University of Tokyo
35	
36	Funding for the MPI Forum meetings was partially supported by award $\#CCF-0816909$
37	and $\#CCF$ -1144042 from the National Science Foundation. In addition, the HDF Group
38	and Sandia National Laboratories provided travel support for one U.S. academic each.
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

## Chapter 1

## Introduction to MPI

#### 1.1 Overview and Goals

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

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

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

A complete list of goals follows.

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

#### **Unofficial Draft for Comment Only**

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 

- 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 messagepassing 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 [51], Express [13], nCUBE's Vertex [47], p4 [8, 9], and PARMACS [5, 10]. Other important contributions have come from Zipcode [54, 55], 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 [61]. 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

#### Unofficial Draft for Comment Only

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.

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

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for <sup>42</sup> both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1" <sup>43</sup> was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for <sup>44</sup> MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done <sup>45</sup> electronically. Both ballots were combined into one document: "Errata for MPI-2", May <sup>46</sup> 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors <sup>47</sup> kept working on new requests for clarification. <sup>48</sup>

#### **Unofficial Draft for Comment Only**

1 2

3

4

5 6

7

8 9

10 11

12

13

14 15

16

17

18

19

20

21

22

23

24 25

26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

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 13 the 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

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 Communications, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations.
- 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.

- 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 Functions, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 16, 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

 $^{24}$ 

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.
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 35 36 37 38 39	Chapter 8, Real-Time MPI, discusses MPI support for real time processing.
40 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. It is similar to the MPI-1 Terms and Conventions chapter but differs in some major and minor ways. Some of the major areas of difference are the naming conventions, some semantic definitions, file objects, Fortran 90 vs Fortran 77, processes, and interaction with signals.

# 2.1 Document Notation

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

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

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

# 2.2 Naming Conventions

In many cases MPI names for C functions are of the form MPI\_Class\_action\_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules.

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.

 $^{24}$ 

a new object, <b>Get</b> retrieves information about an object, <b>Set</b> sets this information, <b>Delete</b> deletes information, <b>Is</b> 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 <b>Class</b> name from the routine and the omission of the <b>Action</b> where one can be
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 <i>references</i> is updated.
<i>Rationale.</i> The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. ( <i>End of rationale.</i> )
MPI's use of IN, OUT and INOUT is intended to indicate to the user how an argument is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., INTENT in Fortran 90 bindings or const in C bindings). For instance, the "constant" MPI_BOTTOM can usually be passed to OUT buffer arguments. Similarly, MPI_STATUS_IGNORE can be passed as the OUT status argument. A common occurrence for MPI functions is an argument that is used as IN by some pro- cesses and OUT by other processes. Such an argument is, syntactically, an INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process. Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process then an arbitrary value can be passed as an argument. Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased with any other argument passed to an MPI procedure. An example of argument aliasing in C appears below. If we define a C procedure like this,
Unofficial Draft for Comment Only
Chometal Drait for Comment Only

# 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,

MPI\_Action\_subset in C and MPI\_ACTION\_SUBSET in Fortran.

2. If the routine is not associated with a class, the name should be of the form

- argument from the
- out value,

paque object (these 25procedure call, then te 26 ugh the handle itself tł 27nat what the handle is 28r29

30  $^{31}$ 

32 33

34ser how an argument 35 e translated directly is 36 onst in C bindings). ir 37  $\mathbf{F}$ UT buffer arguments. 38 $\mathbf{S}$ ment.

39 ed as IN by some pro-40an INOUT argument c  $^{41}$ ll both for input and a 42fc

43ded only by a subset 44en an arbitrary value o 45ca

46 UT cannot be aliased 47argument aliasing in W 48 С

1

2

3

4

5

6 7

8

9

10

11

12

13 14

1516

17

18 19

2021

22 23 $^{24}$ 

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

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

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

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

All MPI functions are first specified in the language-independent notation. Immediately below this, language dependent bindings follow:

- The ISO C version of the function.
- The Fortran version used with USE mpi\_f08.
- The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.

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

#### 2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

- **nonblocking** A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is **started** by the call that initiates it, e.g., MPI\_ISEND. The word complete is used with respect to operations, requests, and communications. An operation completes when the user is allowed to reuse 29 resources, and any output buffers have been updated; i.e. a call to MPI\_TEST will 30 return flag = true. A request is completed by a call to wait, which returns, or a test or get status call which returns flag = true. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is **freed**, and becomes **inactive** if it was persistent. A communication completes when all participating operations complete.
- blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.
- **local** A procedure is local if completion of the procedure depends only on the local executing process.
- **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 communication occurring with another user process.
- collective A procedure is collective if all processes in a process group need to invoke the 4546procedure. A collective call may or may not be synchronizing. Collective calls over 47the same communicator must be executed in the same order by all members of the 48 process group.

## **Unofficial Draft for Comment Only**

1

2

3

4 5

6

7

8 9

10

11

12

13

1415

1617

18

19 20

2122

23 $^{24}$ 

25

26

27

28

 $^{31}$ 

32

33

34

35 36

37

38

39

40 41

42

43

	12	CHAPTER 2.	MPI TERMS AND CONVENTIONS
1 2 3 4 5 6	as MPI_IN MPI_TYF MPI_TYF unname	NT, MPI_FLOAT_INT, or MPI_PACKED) PE_CREATE_F90_INTEGER, MPI_TY PE_CREATE_F90_COMPLEX. The for: d.	PE_CREATE_F90_REAL, or mer are <b>named</b> whereas the latter are
7	derived A der	rived datatype is any datatype that is	not predefined.
8 9 10 11 12 13 14 15 16 17 18 19 20	portable MPI_TYF MPI_TYF Such a da extents o one mem declaratio other han MPI_TYF MPI_TYF ments (e.	datatype using only the type constru PE_VECTOR, MPI_TYPE_INDEXED, MPI_CREATE_SUBARRAY, MPI_TYPE, atatype is portable because all displace of one predefined datatype. Therefore, ory, it will fit the corresponding data ons were used, even if the two system and, if a datatype was constructed us PE_CREATE_HINDEXED_BLOCK, MP PE_CREATE_STRUCT, then the data g., providing padding to meet alignm	MPI_TYPE_CREATE_INDEXED_BLOCK, _DUP, and MPI_TYPE_CREATE_DARRAY. ements in the datatype are in terms of if such a datatype fits a data layout in layout in another memory, if the same s have different architectures. On the ing MPI_TYPE_CREATE_HINDEXED,
21 22 23	used for architect	data layouts on another process, runure.	nning on a processor with a different
24 25			ear to have been created with the same ve the same typemap. Two equivalent

26

# 27

### 2829

30

 $^{31}$ 

2.5

# Data Types

#### 2.5.1 **Opaque Objects**

32 MPI manages system memory that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc. This 33 34memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which 35 exist in user space. MPI procedures that operate on opaque objects are passed handle 36 arguments to access these objects. In addition to their use by MPI calls for object access, 37 handles can participate in assignments and comparisons. 38

datatypes do not necessarily have the same cached attributes or the same names.

39 In Fortran with USE mpi or 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 40objects. With Fortran USE mpi\_f08, the handles are defined as Fortran BIND(C) derived 41 42types that consist of only one element INTEGER :: MPI\_VAL. The internal handle value is identical to the Fortran INTEGER value used in the mpi module and mpif.h. The operators 43 EQ., NE., == and /= are overloaded to allow the comparison of these handles. The type 44names are identical to the names in C, except that they are not case sensitive. For example: 4546

47TYPE, BIND(C) :: MPI\_Comm 48

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 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 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

### Unofficial Draft for Comment Only

 $\frac{4}{5}$ 

 $\overline{7}$ 

 $^{31}$ 

 $46 \\ 47$ 

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). 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;

- 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.
- 42 43

# 2.5.3 State

<sup>44</sup> <sup>45</sup> MPI procedures use at various places arguments with *state* types. The values of such a data <sup>46</sup> type are all identified by names, and no operation is defined on them. For example, the <sup>47</sup> MPI\_TYPE\_CREATE\_SUBARRAY routine has a state argument order with values

<sup>48</sup> MPI\_ORDER\_C and MPI\_ORDER\_FORTRAN.

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

# 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:

	MPI_MAX_PROCESSOR_NAME	
	MPI_MAX_LIBRARY_VERSION_STRING	24
	MPI_MAX_ERROR_STRING	25
	MPI_MAX_DATAREP_STRING	26
	MPI_MAX_DATAKEL_STRING MPI_MAX_INFO_KEY	27
	MPI_MAX_INFO_VAL	28
	MPI_MAX_INFO_VAL MPI_MAX_OBJECT_NAME	29
		30
	MPI_MAX_PORT_NAME	31
	MPI_STATUS_SIZE (Fortran only)	32
	MPI_ADDRESS_KIND (Fortran only)	33
	MPI_COUNT_KIND (Fortran only)	34
	MPI_INTEGER_KIND (Fortran only)	35
	MPI_OFFSET_KIND (Fortran only)	36
	MPI_SUBARRAYS_SUPPORTED (Fortran only)	37
	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)	38
	The constants that cannot be used in initialization expressions or assignments in For-	39
tran	are:	40
	MPI_BOTTOM	41
	MPI_STATUS_IGNORE	42
	MPI_STATUSES_IGNORE	43
	MPI_ERRCODES_IGNORE	44
	MPI_IN_PLACE	45
	MPI_ARGV_NULL	46
	MPI_ARGVS_NULL	40
	MPI_UNWEIGHTED	48
		-40

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

## MPI\_WEIGHTS\_EMPTY

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. 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, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (End of advice to implementors.)

#### 2.5.5 Choice 13

14MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to 15the same routine may pass by reference actual arguments of different types. The mechanism 16for providing such arguments will differ from language to language. For Fortran with the 17include file mpif.h or the mpi module, the document uses <type> to represent a choice variable; with the Fortran mpi\_f08 module, such arguments are declared with the Fortran 192008 + TR 29113 syntax TYPE(\*), DIMENSION(..); for C, we use void \*. 20

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 599. (End of advice to implementors.)

252627

28

29

#### 2.5.6Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the 30 calling program. The datatype of such an argument is MPI\_Aint in C and  $^{31}$ INTEGER (KIND=MPI\_ADDRESS\_KIND) in Fortran. These types must have the same width 32 and encode address values in the same manner such that address values in one language 33 may be passed directly to another language without conversion. There is the MPI constant 34 MPI\_BOTTOM to indicate the start of the address range. 35

36 37

#### 2.5.7 File Offsets

38 For I/O there is a need to give the size, displacement, and offset into a file. These quantities 39 can easily be larger than 32 bits which can be the default size of a Fortran integer. To 40 overcome this, these quantities are declared to be INTEGER (KIND=MPI\_OFFSET\_KIND) in 41 Fortran. In C one uses MPI\_Offset. These types must have the same width and encode 42address values in the same manner such that offset values in one language may be passed 43 directly to another language without conversion.

44 45

#### 2.5.8 Counts 46

47As described above, MPI defines types (e.g., MPI\_Aint) to address locations within memory 48 and other types (e.g., MPI\_Offset) to address locations within files. In addition, some MPI

# **Unofficial Draft for Comment Only**

1

2

3

4

5

6

7

8

9

10

11 12

18

21

22

23

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 593, but that users are recommended not to continue using, since better solutions were provided 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.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

1 Some of the deprecated constructs are now removed, as documented in Chapter 16 on  $^{2}$ page 597. They may still be provided by an implementation for backwards compatibility, 3 but are not required.

4 Table 2.1 shows a list of all of the deprecated and removed constructs. Note that some  $\mathbf{5}$ C typedefs and Fortran subroutine names are included in this list; they are the types of 6 callback functions.

calibacit fulletions.			
Deprecated or removed	deprecated	removed	Replacement
construct	since	since	
MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
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
MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_LB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
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
MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function <sup>2</sup>	MPI-2.0	MPI-3.0	$MPI\_Comm\_errhandler\_function^2$
MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
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>
MPI_NULL_COPY_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_COPY_FN <sup>3</sup>
MPI_NULL_DELETE_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_DELETE_FN <sup>3</sup>
MPI_Copy_function <sup>2</sup>	MPI-2.0		MPI_Comm_copy_attr_function <sup>2</sup>
COPY_FUNCTION <sup>3</sup>	MPI-2.0		COMM_COPY_ATTR_FUNCTION <sup>3</sup>
$MPI_Delete_function^2$	MPI-2.0		$MPI\_Comm\_delete\_attr\_function^2$
DELETE_FUNCTION <sup>3</sup>	MPI-2.0		COMM_DELETE_ATTR_FUNCTION <sup>3</sup>
MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
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>
MPI_COMBINER_STRUCT_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_STRUCT <sup>4</sup>
MPI::	MPI-2.2	MPI-3.0	C language binding
<sup>1</sup> Predefined datatype.			
$^2$ Callback prototype definition.			
<sup>3</sup> Predefined callback routine.			
4 Constant			

- <sup>4</sup> Constant. 38
- Other entries are regular MPI routines. 39
- 40

 $^{41}$ 

4243

Table 2.1: Deprecated and removed constructs

442.6.2 Fortran Binding Issues 45

46Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they 47are 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 48

# **Unofficial Draft for Comment Only**

Fortran 90 or later; it means Fortran 2008 + TR 29113 and later if the mpi\_f08 module is used.

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

All MPI Fortran subroutines have a return code in the last argument. With USE mpi\_f08, this last argument is declared as OPTIONAL, except for user-defined callback functions (e.g., COMM\_COPY\_ATTR\_FUNCTION) and their predefined callbacks (e.g.,

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

Constants representing the maximum length of a string are one smaller in Fortran than in C 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 (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 variables or functions with names beginning with the prefix MPI\_. To support the profiling interface, programs should not declare functions with names beginning with the prefix PMPI\_.

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

### 2.6.4 Functions and Macros

An implementation is allowed to implement MPI\_WTIME, MPI\_WTICK, PMPI\_WTIME, PMPI\_WTICK, and the handle-conversion functions (MPI\_Group\_f2c, etc.) in Section 17.2.4, and no others, as macros in C.

 $^{24}$ 

 $^{31}$ 

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

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

# 2.7 Processes

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

13 This document specifies the behavior of a parallel program assuming that only MPI 14calls are used. The interaction of an MPI program with other possible means of commu-15nication, I/O, and process management is not specified. Unless otherwise stated in the 16specification of the standard, MPI places no requirements on the result of its interaction 17with external mechanisms that provide similar or equivalent functionality. This includes, 18 but is not limited to, interactions with external mechanisms for process control, shared and 19remote memory access, file system access and control, interprocess communication, process 20signaling, and terminal I/O. High quality implementations should strive to make the results 21of such interactions intuitive to users, and attempt to document restrictions where deemed 22necessary. 23

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.

28 29 30

 $^{31}$ 

 $^{24}$ 

25 26

27

# 2.8 Error Handling

32 MPI provides the user with reliable message transmission. A message sent is always received 33 correctly, and the user does not need to check for transmission errors, time-outs, or other 34error conditions. In other words, MPI does not provide mechanisms for dealing with failures 35 in the communication system. If the MPI implementation is built on an unreliable underly-36 ing mechanism, then it is the job of the implementor of the MPI subsystem to insulate the 37 user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, 38 such failures will be reflected as errors in the relevant communication call. Similarly, MPI 39 itself provides no mechanisms for handling processor failures.

40 Of course, MPI programs may still be erroneous. A **program error** can occur when  $^{41}$ an MPI call is made with an incorrect argument (non-existing destination in a send oper-42ation, buffer too small in a receive operation, etc.). This type of error would occur in any 43implementation. In addition, a **resource error** may occur when a program exceeds the 44amount of available system resources (number of pending messages, system buffers, etc.). 45The occurrence of this type of error depends on the amount of available resources in the 46system and the resource allocation mechanism used; this may differ from system to system. 47A high-quality implementation will provide generous limits on the important resources so 48as to alleviate the portability problem this represents.

## Unofficial Draft for Comment Only

1

2

3

4

5 6 7

8 9

10

11

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 19associated with this call will be used to indicate the nature of the error. In a few cases, the 20error 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 21with the ready mode). Such an error must be treated as fatal, since information cannot be 2223returned 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 are available. This is an important point in achieving portability across platforms that provide the same set of services.

#### 2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as write in Fortran and printf and malloc in ISO C) and are executed after

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10 11

12

13

14

15

16

17

18

 $^{24}$ 

25

26

2728

29

30

 $^{31}$ 

32 33

34

35 36 37

38 39

40

41

42

43

4445

4647

<sup>1</sup> MPI\_INIT and before MPI\_FINALIZE operate independently and that their *completion* is <sup>2</sup> 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).

```
8 int rank;
9 MPI_Init((void *)0, (void *)0);
```

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

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

```
<sup>12</sup> MPI_Finalize();
```

<sup>13</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>19</sup> MPI_Comm_rank(MPI_COMM_WORLD, &rank);
<sup>20</sup> printf("Output from task rank %d\n", rank);
```

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

2526

33

34

35

36 37

38 39

21

7

# 2.9.2 Interaction with Signals

<sup>27</sup> MPI does not specify the interaction of processes with signals and does not require that MPI <sup>28</sup> be signal safe. The implementation may reserve some signals for its own use. It is required <sup>29</sup> that the implementation document which signals it uses, and it is strongly recommended <sup>30</sup> that it not use SIGALRM, SIGFPE, or SIGIO. Implementations may also prohibit the use of <sup>31</sup> MPI calls from within signal handlers.

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

# 2.10 Examples

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

```
43
44
```

45

46

47

# Chapter 3

# **Point-to-Point Communication**

#### 3.1Introduction

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

```
20
#include "mpi.h"
                                                                                    21
int main( int argc, char *argv[])
                                                                                    22
{
                                                                                    23
  char message[20];
  int myrank;
 MPI_Status status;
 MPI_Init( &argc, &argv );
                                                                                    27
 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
                                                                                    28
  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();
  return 0;
}
```

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

7

9

1516

17

18

19

 $^{24}$ 

25

26

38

39

40

41

46

47

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

11 The next sections describe the blocking send and receive operations. We discuss send, 12receive, blocking communication semantics, type matching requirements, type conversion in 13 heterogeneous environments, and more general communication modes. Nonblocking com-14munication is addressed next, followed by probing and canceling a message, channel-like 15constructs and send-receive operations, ending with a description of the "dummy" process, 16MPI\_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 28	IN	count	number of elements in send buffer (non-negative integer)
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 34	IN	comm	communicator (handle)

int MPI\_Send(const void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
38
39
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
40
         INTEGER, INTENT(IN) :: count, dest, tag
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
         INTEGER, OPTIONAL, INTENT(OUT) ::
43
```

```
ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
45
         <type> BUF(*)
46
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
47
```

```
48
          The blocking semantics of this call are described in Section 3.4.
```

## **Unofficial Draft for Comment Only**

1718

19 20

21

22 23 $^{24}$ 

25

35

36

37

## 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

### Unofficial Draft for Comment Only

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

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 datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
L	MPI_SHORT	signed short int
i	MPI_INT	signed int
	MPI_LONG	signed long int
	MPI_LONG_LONG_INT	signed long long int
3	MPI_LONG_LONG (as a synonym)	signed long long int
	MPI_SIGNED_CHAR	signed tong tong int
)		(treated as integral value)
1	MPI_UNSIGNED_CHAR	unsigned char
2		(treated as integral value)
3	MPI_UNSIGNED_SHORT	unsigned short int
4	MPI_UNSIGNED	unsigned int
5	MPI_UNSIGNED_LONG	0
6		unsigned long int
7	MPI_UNSIGNED_LONG_LONG	unsigned long long int float
8	MPI_FLOAT	
9		double
0	MPI_LONG_DOUBLE	long double
	MPI_WCHAR	wchar_t
1		(defined in <stddef.h>)</stddef.h>
2		(treated as printable character)
3	MPI_C_BOOL	Bool
4	MPI_INT8_T	int8_t
5	MPI_INT16_T	int16_t
6	MPI_INT32_T	int32_t
7	MPI_INT64_T	int64_t
3	MPI_UINT8_T	uint8_t
9	MPI_UINT16_T	uint16_t
)	MPI_UINT32_T	uint32_t
L	MPI_UINT64_T	uint64_t
2	MPI_C_COMPLEX	float _Complex
3	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
1	MPI_C_DOUBLE_COMPLEX	double _Complex
5	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
3	MPI_BYTE	
7	MPI_PACKED	
3		
9	Table 3.2: Predefined MPI datatypes co	prresponding to C datatypes
0	rable 5.2. i redefined Mi i datatypes (	morphume to C datatypes
1		
2	INTEGER (KIND=MPI_ADDRESS_KIND), INTEGER (	-
3	INTEGER (KIND=MPI_COUNT_KIND). This is described	
4	handles are available in all language bindings. See	* 0
5	and 660 for information on interlanguage communi	
6	If there is an accompanying $C++$ compiler the transmission of $C+++$ compiler the transmission of $C++++++++++++++++++++++++++++++++++++$	hen the datatypes in Table $3.4$ are als

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,...,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.

41 The communicator also specifies the set of processes that share this communication 42context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is  $0, \ldots, n-1$ , where n is the number of 4344processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

46A predefined communicator MPI\_COMM\_WORLD is provided by MPI. It allows com-47munication with all processes that are accessible after MPI initialization and processes are 48 identified by their rank in the group of MPI\_COMM\_WORLD.

#### **Unofficial Draft for Comment Only**

151617

18

19

20

27

2829

30

 $^{31}$ 

32 33

34

35

36 37

38 39

40

1 Advice to users. Users that are comfortable with the notion of a flat name space 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 29 IN message tag or MPI\_ANY\_TAG (integer) tag 30 IN comm communicator (handle)  $^{31}$ 32 OUT status status object (Status) 33 34int 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), 47IERROR 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 2728wildcard 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 argument 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 intercommunicators). Thus, the range of valid values for the source argument is  $\{0, \dots, n-1\}$  $1 \cup \{ MPI_ANY_SOURCE \}, where n is the number of processes in this group.$ 

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

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4 5

6 7

8

9

10 11

12

13

14

15

1617

18

19

20

21

22

 $^{24}$ 25

26

 $^{31}$ 

32

33

34

35

36

37

3839

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 81.

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,

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 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 650.

*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.)

# Unofficial Draft for Comment Only

2 3 4

5 6 7

8

32

33

34

35 36

37

38

### 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

 $^{31}$ 

 $44 \\ 45$ 

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 10 described in this section. 11

1213

1

2

3

4

5

6

7 8

9

#### 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 16is returned. An object of type MPI\_Status is not an MPI opaque object; its structure 17is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, 18 application programs are constructed so that it is unnecessary for them to examine the 19 status fields. In these cases, it is a waste for the user to allocate a status object, and it is 20particularly wasteful for the MPI implementation to fill in fields in this object. 21

To cope with this problem, there are two predefined constants, MPI\_STATUS\_IGNORE 22 and MPI\_STATUSES\_IGNORE, which when passed to a receive, probe, wait, or test function, 23inform 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

- 47
- 48

#### 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.

### **Unofficial Draft for Comment Only**

5

6 7

8 9

10 11

12

13

1415

16

17

18

1 2

19 2021

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
8
     be 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 bound 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.

# Unofficial Draft for Comment Only

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 MDI_RECV(b(6:10) E MDI_CHARACTER 0 tag comm status ierr)
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>

### Unofficial Draft for Comment Only

typed value is transferred across environments that use different representations for the
 datatype of this value. MPI does not specify rules for representation conversion. Such
 conversion is expected to preserve integer, logical or character values, and to convert a
 floating point value to the nearest value that can be represented on the target system.

<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  $\geq 10$ . 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. 22While 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 647.

### Unofficial Draft for Comment Only

# 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 44 45 46 47 48

### Unofficial Draft for Comment Only

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-7 fined. 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 13 posted), 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-24ger) 25datatype of each send buffer element (handle) IN datatype 2627dest IN 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. 45 46 47 48

MPI	_SSEND (buf, count, datatype, de	st, tag, comm)	1	
IN	buf	initial address of send buffer (choice)	2 3	
IN	count	number of elements in send buffer (non-negative integer)	3 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	-	communicator (handle)	10	
			11	
int	MPI_Ssend(const void* buf, int tag, MPI_Comm	int count, MPI_Datatype datatype, int dest, comm)	12 13 14	
MPI_	Ssend(buf, count, datatype,	<pre>dest, tag, comm, ierror) BIND(C)</pre>	15	
	TYPE(*), DIMENSION(), INT		16	
INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype				
	TYPE(MPI_Datatype), INTENI( TYPE(MPI_Comm), INTENT(IN)		18	
	INTEGER, OPTIONAL, INTENT(O		19 20	
NDT			20	
MP1_	_SSEND(BUF, COUNT, DATATYPE, <type> BUF(*)</type>	DEST, TAG, CUMM, IERRUR)	22	
	INTEGER COUNT, DATATYPE, DE	ST. TAG. COMM. TERROR	23	
			24	
	Send in synchronous mode.		25	
			26 27	
MPI_RSEND (buf, count, datatype, dest, tag, comm)				
IN	buf	initial address of send buffer (choice)	29	
IN	count	number of elements in send buffer (non-negative integer)	30 31	
IN	datatype	datatype of each send buffer element (handle)	32	
IN		rank of destination (integer)	33 34	
IN		message tag (integer)	35	
	5	0 0 ( 0 )	36	
IN	comm	communicator (handle)	37	
int	MPI_Rsend(const void* buf, int tag, MPI_Comm	int count, MPI_Datatype datatype, int dest, comm)	38 39 40	
мрт	MDI Reand(buf count datatupe doct the comm isomer) PIND(C)			
· · · · · _	<pre>MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: buf</pre>			
	INTEGER, INTENT(IN) :: count, dest, tag			
	TYPE(MPI_Datatype), INTENT(	-	44	
	TYPE(MPI_Comm), INTENT(IN)		45 46	
	INTEGER, OPTIONAL, INTENT(O	UT) :: ierror	40 47	
MPI_	IPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)			

Unofficial Draft for Comment Only

1	<type> BUF(*)</type>		
2	INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR		
3	Send in ready mode.		
4 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 <b>blocking</b> : 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). In a multithreaded implementation of MPI, the system may de-schedule a thread that		
9	is blocked on a send or receive operation, and schedule another thread for execution in		
10 11	the same address space. In such a case it is the user's responsibility not to modify a		
12	communication buffer until the communication completes. Otherwise, the outcome of the		
13	computation is undefined.		
14	Advice to implementors. Since a synchronous send cannot complete before a matching		
15 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 19	for standard sends. The programmer can signal his or her preference for blocking t sender until a matching receive occurs by using the synchronous send mode.		
20 21	A possible communication protocol for the various communication modes is outlined below.		
22 23	ready send: The message is sent as soon as possible.		
24 25 26	this request. When a matching receive is posted, the receiver sends back a permission		
27 28	<b>standard send</b> : First protocol may be used for short messages, and second protocol for long messages.		
29 30 31	<sup>30</sup> buffered send: The sender copies the message into a buffer and then sends it with nonblocking send (using the same protocol as for standard send).		
32 33	Additional control messages might be needed for flow control and error recovery. (		
34 35	Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.		
36 37 38	A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.		
39	In a multithreaded environment, the execution of a blocking communication should		
40 41	block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. ( <i>End of advice to implementors.</i> )		
42			
43 44	3.5 Semantics of Point-to-Point Communication		
45 46 47	<sup>15</sup> A valid MPI implementation guarantees certain general properties of point-to-point com <sup>16</sup> munication, which are described in this section.		

**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.

An example of two, intertwined matching pairs. Example 3.6

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

1

2

3

4

56

 $\overline{7}$ 

8 9

10

11

12

13

1415

16

17

18

1920

21

22

23

 $^{24}$ 

25

26

27

2829

30

3132

33

34

35

36

37 38

> 39 40

41

44

45

46

47

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.

10 Fairness 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 38 and 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 40 will 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

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated
 by the examples below.

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

47 48

```
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
                                                                                       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
ELSE IF (rank.EQ.1) THEN
                                                                                       34
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
                                                                                       35
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
                                                                                       36
END IF
```

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

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

#### Unofficial Draft for Comment Only

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 not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

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

 $^{24}$ 

# 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
Descrides to MDL a buffer in the user's memory to be used for buffering output
```

Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can
be attached to a process at a time. In C and C++, buffer 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 628).

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 34 differently: 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. 36 In Fortran with the mpi module or mpif.h, the type of the buffer\_addr argument is 37 wrongly defined and the argument is therefore unused. In Fortran with the mpi\_f08 38 module, the address of the buffer is returned as TYPE(C\_PTR), see also Example 8.1 39 on page 341 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

associated with the process.

<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.)

#### 19 20 21

22

12

13

14

15 16

17

18

# 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

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 20that the 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 send-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

### **Unofficial Draft for Comment Only**

1

2

3 4 5

6  $\overline{7}$ 

8 9

10 11

12

13

1415

16

17

18

19

 $^{24}$ 

2526

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.

39

34

35

- 40 41
- 42
- 43
- 44 45
- 15
- 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 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_		t count, MPI_Datatype datatype, int dest, mm, MPI_Request *request)	13 14 15		
TYPE INTE TYPE TYPE TYPE	• =	) :: datatype comm ) :: request	<ol> <li>16</li> <li>17</li> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> </ol>		
<pre>MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>					
MPI_IBSE	ND(buf, count, datatype, dest,	tag, comm, request)	29 30		
IN	buf	initial address of send buffer (choice)	31		
IN	count	number of elements in send buffer (non-negative inte- ger)	32 33 34		
IN	datatype	datatype of each send buffer element (handle)	35		
IN	dest	rank of destination (integer)	36		
IN	tag	message tag (integer)	37 38		
IN	comm	communicator (handle)	39		
OUT	request	communication request (handle)	40 41		
int MPI_1		nt count, MPI_Datatype datatype, int dest, mm, MPI_Request *request)	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
       IN
                 datatype
                                             datatype of each send buffer element (handle)
16
17
                 dest
       IN
                                             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_IRSEI	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)	3 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_I		nt count, MPI_Datatype datatype, int dest, mm, MPI_Request *request)	13 14 15
TYPE( INTEG TYPE( TYPE( TYPE(	• -	) :: datatype comm ) :: request	16 17 18 19 20 21 22 23
<type INTEG</type 	> BUF(*)	DEST, TAG, COMM, REQUEST, IERROR) , TAG, COMM, REQUEST, IERROR nd.	24 25 26 27 28 29
MPI_IREC	V (buf, count, datatype, source	e, tag, comm, request)	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
<pre>4 int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,</pre>			
INTEGER, INTENT(IN) :: count, source, tag47TYPE(MPI_Datatype), INTENT(IN) :: datatype48			

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 10 handle (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 17.1.10-17.1.20, 20especially in Sections 17.1.12 and 17.1.13 on pages 628-630 about "Problems Due to 2122Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", 23and in Sections 17.1.16 to 17.1.19 on pages 633 to 643 about "Optimization Problems", 24"Code Movements and Register Optimization", "Temporary Data Movements" and "Per-25manent Data Movements". (End of advice to users.) 2627Communication Completion 3.7.3 28The functions MPI\_WAIT and MPI\_TEST are used to complete a nonblocking communica-29tion. The completion of a send operation indicates that the sender is now free to update the 30 locations in the send buffer (the send operation itself leaves the content of the send buffer  $^{31}$ unchanged). It does not indicate that the message has been received, rather, it may have 32 been buffered by the communication subsystem. However, if a synchronous mode send was 33 used, the completion of the send operation indicates that a matching receive was initiated, 34 and that the message will eventually be received by this matching receive. 35 The completion of a receive operation indicates that the receive buffer contains the 36 received message, the receiver is now free to access it, and that the status object is set. It 37 does not indicate that the matching send operation has completed (but indicates, of course, 38 that the send was initiated). 39 We shall use the following terminology: A **null** handle is a handle with value 40 MPI\_REQUEST\_NULL. A persistent request and the handle to it are **inactive** if the re-41 quest is not associated with any ongoing communication (see Section 3.9). A handle is 42active if it is neither null nor inactive. An empty status is a status which is set to 43 return tag = MPI\_ANY\_TAG, source = MPI\_ANY\_SOURCE, error = MPI\_SUCCESS, and is 44 also internally configured so that calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and 45

<sup>46</sup> MPI\_GET\_ELEMENTS\_X return count = 0 and MPI\_TEST\_CANCELLED returns false. We <sup>47</sup> set a status variable to empty when the value returned by it is not significant. Status is set <sup>48</sup> in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI\_WAIT, MPI\_TEST, or any of the other derived functions (MPI\_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI\_ERR\_IN\_STATUS; and the returned status can be queried by the call MPI\_TEST\_CANCELLED.

Error codes belonging to the error class MPI\_ERR\_IN\_STATUS should be returned only by the MPI completion functions that take arrays of MPI\_Status. For the functions MPI\_TEST, MPI\_TESTANY, MPI\_WAIT, and MPI\_WAITANY, which return a single MPI\_Status value, the normal MPI error return process should be used (not the MPI\_ERROR field in the MPI\_Status argument).

MPI\_WAIT(request, status)

INOUT OUT	•	request (handle) status object (Status)
int MPI_Wa	ait(MPI_Request *request,	MPI_Status *status)
TYPE(N TYPE(N	request, status, ierror) MPI_Request), INTENT(INOU MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)	T) :: request
	REQUEST, STATUS, IERROR) ER REQUEST, STATUS(MPI_ST	ATUS_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.*)

```
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
20
     of request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call
21
     returns flag = false if the operation identified by request is not complete. In this case, the
22
     value of 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}
           Advice to users.
                              The use of the nonblocking MPI_TEST call allows the user to
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
                       Simple usage of nonblocking operations and MPI_WAIT.
     Example 3.11
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
INOUT request c	communication request (handle)	6 7
		8
<pre>int MPI_Request_free(MPI_Request *re</pre>	equest)	9
MPI_Request_free(request, ierror) BI	IND (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
a 0	ion and set request to MPI_REQUEST_NULL. An	16
request will be deallocated only after its co	with the request will be allowed to complete. The	17 18
request will be deallocated only after its et	mpicuon.	19
Rationale. The MPI_REQUEST_FR	EE mechanism is provided for reasons of perfor-	20
mance and convenience on the sendir	ng side. (End of rationale.)	21
		22
-	reed by a call to MPI_REQUEST_FREE, it is not	23
	ompletion of the associated communication with Also, if an error occurs subsequently during the	24
	t be returned to the user — such an error must	25
	ve request should never be freed as the receiver	26
	eceive has completed and the receive buffer can	27 28
be reused. (End of advice to users.)		29
		30
<b>Example 3.12</b> An example using MPI.	REQUEST EREE	31
Example 5.12 All example using Wiri		32
CALL MPI_COMM_RANK(MPI_COMM_WORLD, 1	rank, ierr)	33
IF (rank.EQ.0) THEN		34
DO i=1, n		35
	_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	36
CALL MPI_REQUEST_FREE(req, ie)		37 38
	REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	39
CALL MPI_WAIT(req, status, ie) END DO	(1)	40
ELSE IF (rank.EQ.1) THEN		41
	AL, 0, 0, MPI_COMM_WORLD, req, ierr)	42
CALL MPI_WAIT(req, status, ierr)	-	43
DO I=1, n-1		44
	I_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	45
CALL MPI_REQUEST_FREE(req, ie		46
	_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	47 48
CALL MPI_WAIT(req, status, ie	err)	40

**Unofficial Draft for Comment Only** 

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
     3.7.4
            Semantics of Nonblocking Communications
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
37
     Example 3.14
                        An illustration of progress semantics.
38
     CALL MPI_COMM_RANK(comm, rank, ierr)
39
     IF (RANK.EQ.0) 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)

IN	count	list length (non-negative integer)	2
INOUT	array_of_requests	array of requests (array of handles)	2
OUT	index	index of handle for operation that completed (integer)	2
OUT	status	status object (Status)	2
			2
int MPI_W	<pre>int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,</pre>		
	MPI_Status *status)		2
			3

```
MPI_Waitany(count, array_of_requests, index, status, ierror) BIND(C)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: 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

#### Unofficial Draft for Comment Only

```
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
46
      last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an
47
      array containing one active entry is equivalent to MPI_TEST.
48
```

MPI_WAITALL( count, array_of_requests, array_of_statuses)			1
IN count lists length (non-negative integer)			
		,	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
······································			10
	INTEGER, INTENT(IN) :: count		
			12
			13
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
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	s until all communication ope	erations associated with active handles in the list	19

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.*)

 $^{24}$ 

```
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 as errors in
32
     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)	38
INOUT	array_of_requests	array of requests (array of handles)	39
OUT	outcount	number of completed requests (integer)	40
OUT	array_of_indices	array of indices of operations that completed (array of	41
001	array_or_mulces	· - · · ·	42
		integers)	43
OUT	array_of_statuses	array of status objects for operations that completed	44
		(array of Status)	45
			46

### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

1	<pre>MPI_Status array_of_statuses[])</pre>
2 3	MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
4	array_of_statuses, ierror) BIND(C)
5	INTEGER, INTENT(IN) :: incount
6	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount) INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
7	TYPE(MPI_Status) :: array_of_statuses(*)
8 9	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9 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.
18	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 fairness requirement: If a request for
21	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 succeed, unless the send is satisfied by another receive; and similarly for send requests.
23 24	Errors that occur during the execution of MPI_TESTSOME are handled as for
24 25	MPI_WAITSOME.
26	
27	Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use
28 29	of MPI_TESTANY. The former returns information on all completed communications, 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 34	that completed. If a call to MPI_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. ( <i>End of advice to users.</i> )
34 35	while requests from another chemi arways sheak in hist. ( <i>Linu of autiet to users.</i> )
36	Advice to implementors. MPI_TESTSOME should complete as many pending com-
37	munications as possible. (End of advice to implementors.)
38	
39	<b>Example 3.15</b> Client-server code (starvation can occur).
40	• • • • • • • • • • • • • • • • • • • •
41 42	
42	CALL MPI_COMM_SIZE(comm, size, ierr) 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)
                                                                                      11
       END DO
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.

Unofficial Draft for Comment Only

41

42

43

44

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

```
2
       IN
                 request
                                             request (handle)
3
       OUT
                                             boolean flag, same as from MPI_TEST (logical)
                 flag
4
5
       OUT
                 status
                                             status object if flag is true (Status)
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 21if the operation is not complete. 22

One is allowed to call MPI\_REQUEST\_GET\_STATUS with a null or inactive request 23argument. In such a case the operation returns with flag=true and empty status.  $^{24}$ 

2526

27

#### 3.8 Probe and Cancel

28The MPI\_PROBE, MPI\_IPROBE, MPI\_MPROBE, and MPI\_IMPROBE operations allow in-29coming messages to be checked for, without actually receiving them. The user can then 30 decide how to receive them, based on the information returned by the probe (basically, the  $^{31}$ information returned by status). In particular, the user may allocate memory for the receive 32 buffer, according to the length of the probed message.

33 The MPI\_CANCEL operation allows pending communications to be cancelled. This is 34required for cleanup. Posting a send or a receive ties up user resources (send or receive 35 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)

41				
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 81.

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
OUT	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

#### **Unofficial Draft for Comment Only**

```
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
                         CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
     100
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

#### Unofficial Draft for Comment Only

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

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.

```
6
     MPI_IMPROBE(source, tag, comm, flag, message, status)
7
8
       IN
                                            rank of source or MPI_ANY_SOURCE (integer)
                source
9
       IN
                                            message tag or MPI_ANY_TAG (integer)
                tag
10
       IN
                comm
                                            communicator (handle)
11
12
       OUT
                flag
                                            flag (logical)
13
       OUT
                 message
                                            returned message (handle)
14
       OUT
                status
                                            status object (Status)
15
16
17
     int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
18
                    MPI_Message *message, MPI_Status *status)
19
     MPI_Improbe(source, tag, comm, flag, message, status, ierror) BIND(C)
20
         INTEGER, INTENT(IN) :: source, tag
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, INTENT(OUT) :: flag
23
         TYPE(MPI_Message), INTENT(OUT) :: message
^{24}
         TYPE(MPI_Status) :: status
25
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
26
27
     MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
          INTEGER SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS(MPI_STATUS_SIZE),
28
29
         IERROR
30
```

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

A matched receive (MPI\_MRECV or MPI\_IMRECV) executed with the message handle will receive the message that was matched by the probe. Unlike MPI\_IPROBE, no other probe or receive operation may match the message returned by MPI\_IMPROBE. 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

synchronou There which has MPI_MESSA A mat MPI_MESSA = 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 fo tching probe with MPI_PROC AGE_NO_PROC, and the statu Y_TAG, and count = 0; see Se RECV with MPI_MESSAGE_NO male. MPI_MESSAGE_NO	n has started to receive the message sent by the ge: MPI_MESSAGE_NO_PROC, which is a message e process. The predefined constant r invalid message handles. C_NULL as source returns flag = true, message = s object returns source = MPI_PROC_NULL, tag ction 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 woid 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	-		20
	comm	communicator (handle)	21
OUT	message	returned message (handle)	22
OUT	status	status object (Status)	23 24
int MPI_M	probe(int source, int tag MPI_Status *status)	g, MPI_Comm comm, MPI_Message *message,	25 26 27
INTEG TYPE( TYPE( TYPE(	e(source, tag, comm, mess ER, INTENT(IN) :: source MPI_Comm), INTENT(IN) :: MPI_Message), INTENT(OUT) MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)	comm :: message	28 29 30 31 32 33
MPI MPROB	E(SOURCE, TAG, COMM, MESS	AGE. STATUS. IERROR)	34
		SAGE, STATUS(MPI_STATUS_SIZE), IERROR	35 36
only after a The in	a matching message has been	$BE \ \mathrm{and} \ MPI\_IMPROBE \ \mathrm{needs} \ \mathrm{to} \ \mathrm{guarantee} \ \mathrm{progress}$	37 38 39 40
3.8.3 Ma	tched Receives		$41 \\ 42$
			43
		RECV receive messages that have been previously	44
matched by	y a matching probe (Section $3$	.0.2).	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) 1213 MPI\_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 34 handle. 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 44 4546 47 48

	ECV(buf, count, datatype	_ , ,	1 2
OUT	buf	initial address of receive buffer (choice)	3
IN	count	number of elements in receive buffer (non-negative in-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
tut NDT	T		10
int MPI_		count, MPI_Datatype datatype,	11 12
	MFI_Message *me	ssage, MPI_Request *request)	12
	-	vpe, message, request, ierror) BIND(C)	14
	(*), DIMENSION(), A	ASYNCHRONOUS :: buf	15
			16
	(MPI_Datatype), INTEN	* -	17
	(MPI_Message), INTEN (MPI_Request), INTEN		18
	GER, OPTIONAL, INTENI	1	19
			20
		YPE, MESSAGE, REQUEST, IERROR)	21
• 1	e> BUF(*)		22
	GER CUUNI, DAIAIYPE,	MESSAGE, REQUEST, IERROR	23 24
		king variant of MPI_MRECV and starts a nonblocking	25
	0	pletion semantics are similar to MPI_IRECV as described	26
		this function, the message handle is set to	27
	SAGE_NULL.	MDI MESSAGE NO DDOG the message and the	28
		MPI_MESSAGE_NO_PROC as the message argument, the quest object which, when completed, will yield a status	29
	_	C_NULL, tag = MPI_ANY_TAG, and count = 0, as if a	30
-		issued (see Section 3.11). A call to MPI_IMRECV with	31
	SAGE_NULL is erroneous.		32
			33 34
	-	reception of a matched message is started with	35
$MPI\_IMRECV,$ then it is possible to cancel the returned request with $MPI\_CANCEL.$ If			36
MPI_CANCEL succeeds, the matched message must be found by a subsequent message			
-		PROBE, MPI_MPROBE, or MPI_IMPROBE), received by	38
		on or cancelled by the sender. See Section 3.8.4 for details neellation of operations initiated with MPI_IMRECV may	39
abor		incontation of operations infitiated with with Livin LCV Illay	40

3.8.4 Cancel

fail. (End of advice to implementors.)

MPI_CANO	CEL(request)	
IN	request	communication request (handle)

# **Unofficial Draft for Comment Only**

 $\frac{41}{42}$ 

```
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)
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 16 MPI\_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

 $_{33}$  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)

34 35

36

37

38

43	IN	status	status object (Sta	itus)
45	OUT	flag	(logical)	
46 47 48	int MPI_	Test_cancelled(const	MPI_Status *status,	int *flag)

```
MPI_Test_cancelled(status, flag, ierror) BIND(C)
    TYPE(MPI_Status), INTENT(IN) :: status
    LOGICAL, INTENT(OUT) :: flag
    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 "half-channel." 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.

 $^{24}$ 

 $^{31}$ 

1 MPI\_SEND\_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 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 communication request (handle) request 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 22INTEGER, 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 datatype IN 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(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request	1 2 3
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	5 6
	<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>	7 8
	Creates a persistent communication request for a buffered mode send.	9
		10
		11
MPI_	SSEND_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
		18
IN	tag message tag (integer)	19
IN	comm communicator (handle)	20
OU	T request communication request (handle)	21 22
	<pre>MPI_Ssend_init(const void* buf, int count, MPI_Datatype datatype,</pre>	23 24 25
MPI_	Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)	26
	BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	27 28
	INTEGER, INTENT(IN) :: count, dest, tag	28 29
	TYPE(MPI_Datatype), INTENT(IN) :: datatype	30
	TYPE(MPI_Comm), INTENT(IN) :: comm	31
	TYPE(MPI_Request), INTENT(OUT) :: request	32
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
	SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*)</type>	34 35
	INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	36 37
	Creates a persistent communication object for a synchronous mode send operation.	38
		39
		40
		41
		42
		43 44
		45
		46
		47

```
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
                                             communication request (handle)
                 request
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
                                             number of elements received (non-negative integer)
                 count
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
	MPI_Request), INTENT(OUT)	1	2
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	3
MPT RECV	TNIT (BUF. COUNT. DATATYPE	E, SOURCE, TAG, COMM, REQUEST, IERROR)	4
	> BUF(*)	_,,,,,,,,,	5
• -		CE, TAG, COMM, REQUEST, IERROR	6
			7
	*	request for a receive operation. The argument buf	8 9
	_	permission to write on the receive buffer by passing	9 10
0	ent to MPI_RECV_INIT.		11
		t is inactive after it was created — no active com-	12
	n is attached to the request.	that uses a persistent request is initiated by the	13
	IPI_START.	that uses a persistent request is initiated by the	14
			15
			16
MPI_STAR	RT(request)		17
INOUT	request	communication request (handle)	18
			19
int MPT S	tart(MPI_Request *request	-)	20
			21
	(request, ierror) BIND(C)		22
	MPI_Request), INTENT(INOU	-	23
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	24
MPI_START	(REQUEST, IERROR)		25
	ER REQUEST, IERROR		26
			27
	S , . ,	e returned by one of the previous five calls. The	28 29
	-	he request becomes active once the call is made. ly mode, then a matching receive should be posted	29 30
	-	ation buffer should not be modified after the call,	31
	he operation completes.	ation build should not be modified after the carl,	32
		tics to the nonblocking communication operations	33
		to MPI_START with a request created by	34
	,	n in the same manner as a call to MPI_ISEND; a	35
		ted by MPI_BSEND_INIT starts a communication	36
	e manner as a call to MPI_IB		37
			38
		<b>`</b>	39
MPI_STAR	TALL(count, array_of_request:	5)	40
IN	count	list length (non-negative integer)	41
INOUT	array_of_requests	array of requests (array of handle)	42
		· · · · /	43
int MPT S	tartall(int count, MPT Re	equest array_of_requests[])	44
			45
	all(count, array_of_reque	ests, ierror) BIND(C)	46
INTEG			
	ER, INTENT(IN) :: count		47 48
TYPE(	-	<pre>JT) :: array_of_requests(count)</pre>	47 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 17 to 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 19be invoked in a sequence of the form, 2021Create (Start Complete)\* Free 22 23where \* indicates zero or more repetitions. If the same communication object is used in  $^{24}$ several concurrent threads, it is the user's responsibility to coordinate calls so that the 25correct sequence is obeyed. 26A send operation initiated with MPI\_START can be matched with any receive operation 27and, likewise, a receive operation initiated with MPI\_START can receive messages generated 28by any send operation. 29 30 Advice to users. To prevent problems with the argument copying and register opti- $^{31}$ mization done by Fortran compilers, please note the hints in Sections 17.1.10-17.1.20, 32 especially in Sections 17.1.12 and 17.1.13 on pages 628-630 about "Problems Due to 33 Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", 34 and in Sections 17.1.16 to 17.1.19 on pages 633 to 643 about "Optimization Problems". 35 "Code Movements and Register Optimization", "Temporary Data Movements" and "Per-36 manent Data Movements". (End of advice to users.) 37 38 Send-Receive 3.10 39 40 The send-receive operations combine in one call the sending of a message to one desti-41 nation and the receiving of another message, from another process. The two (source and 42destination) are possibly the same. A send-receive operation is very useful for executing 43 a shift operation across a chain of processes. If blocking sends and receives are used for 44 such a shift, then one needs to order the sends and receives correctly (for example, even 45processes send, then receive, odd processes receive first, then send) so as to prevent cyclic 46dependencies that may lead to deadlock. When a send-receive operation is used, the com-47munication subsystem takes care of these issues. The send-receive operation can be used 48

in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

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

MPI\_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, 9 source, recvtag, comm, status) 10 11 IN sendbuf initial address of send buffer (choice) 12IN sendcount number of elements in send buffer (non-negative inte-13 ger) 14IN sendtype type of elements in send buffer (handle) 1516dest rank of destination (integer) IN 17IN sendtag send tag (integer) 18 OUT recvbuf initial address of receive buffer (choice) 19 IN recvcount number of elements in receive buffer (non-negative in-2021teger) 22 IN type of elements in receive buffer (handle) recvtype 23rank of source or MPI\_ANY\_SOURCE (integer) IN source 2425IN recvtag receive tag or MPI\_ANY\_TAG (integer) 26IN comm communicator (handle) 27OUT status object (Status) status 2829

MPI\_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> SENDBUF(\*), RECVBUF(\*)

1 2

3

4

5 6

7 8

30

31

32

33 34

35

36

37

38

39

40

41

42

43

44

45

46

47

	80	CHAPT	FER 3.	POINT-TO-POINT COMMUNICATION
<ul> <li>INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTY</li> <li>SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR</li> </ul>				
3 4 5 6 7 8 9 10	Execute a blocking send and receive operation. Both send and receive use the same 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 two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.			
11 12	MPI_SEND	RECV_REPLACE(buf, count, tus)	datatyp	e, dest, sendtag, source, recvtag, comm, sta-
13 14	INOUT	buf	initial	address of send and receive buffer (choice)
15 16	IN	count		er of elements in send and receive buffer (non- ve integer)
17	IN	datatype	type o	f elements in send and receive buffer (handle)
18 19	IN	dest	rank o	f destination (integer)
20	IN	sendtag	send n	nessage tag (integer)
21	IN	source	rank o	f source or MPI_ANY_SOURCE (integer)
22	IN	recvtag	receive	e message tag or $MPI_ANY_TAG$ (integer)
23 24	IN	comm	comm	unicator (handle)
25	OUT	status	status	object (Status)
26 27 28 29 30	int MPI_S	-		<pre>count, MPI_Datatype datatype, source, int recvtag, MPI_Comm comm,</pre>
31 32		comm, status, ierror	) BIND	oe, dest, sendtag, source, recvtag, (C)
33	TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag			
$\frac{34}{35}$	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
36	TYPE(MPI_Comm), INTENT(IN) :: comm			
37		MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)	:: i	error
38 39				PE, DEST, SENDTAG, SOURCE, RECVTAG,
40		COMM, STATUS, IERROR		
41	• 1	> BUF(*)		
42 43		ER COUNT, DATATYPE, DEST, S(MPI_STATUS_SIZE), IERRC		AG, SOURCE, RECVTAG, COMM,
44		-		
45		te a blocking send and receive eive, so that the message sent		same buffer is used both for the send and
46 47		,	-	
48		nt. (End of advice to implement		nediate buffering is needed for the "replace"

#### 3.11 Null Processes

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

The special value MPI\_PROC\_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI\_PROC\_NULL has no effect. A send to MPI\_PROC\_NULL succeeds and returns as soon as possible. A receive from MPI\_PROC\_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI\_PROC\_NULL is executed then the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG and count = 0. A 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 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 as source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0.

 $\overline{7}$ 

## Chapter 4

## Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 25 and in Section 3.3 Data Type Matching and Data Conversion 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.

## 4.1 Derived Datatypes

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

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

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

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

• A sequence of basic datatypes

 $^{24}$ 

 $46 \\ 47$ 

• A sequence of integer (byte) displacements

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

8

Let

1

2

3

4

5

6

7

9

10 11

12

13

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

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

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

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

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

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

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

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

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

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

then

39 40 41

42 43 44

37

38

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

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

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

<sup>45</sup> If  $type_i$  requires alignment to a byte address that is a multiple of  $k_i$ , then  $\epsilon$  is the least <sup>46</sup> non-negative increment needed to round extent(Typemap) to the next multiple of  $\max_i k_i$ . <sup>47</sup> In Fortran, it is implementation dependent whether the MPI implementation computes <sup>48</sup> the alignments  $k_i$  according to the alignments used by the compiler in common blocks,

SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE nor BIND(C). The complete definition of **extent** is given in Section 4.1.6 on page 104.

**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 104 and in Section 17.1.15 on page 631. (*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), where ever 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)	36
IN	oldtype	old datatype (handle)	37
			38
Ol	IT newtype	new datatype (handle)	39
			40
int	MPI_Type_contiguous(int coun	t, MPI_Datatype oldtype,	41
	MPI_Datatype *newty	ype)	42
мрт	Type contiguous(count oldty	pe, newtype, ierror) BIND(C)	43
· · · · -	INTEGER, INTENT(IN) :: coun		44
	TYPE(MPI_Datatype), INTENT(I		45
	TYPE(MPI_Datatype), INTENT(C	• -	46
	INTEGER, OPTIONAL, INTENT(OU	01	47
	INTEGER, OF ITOWAL, INTENT(OU	(1) IEIIOI	48

 $\overline{7}$ 

 $^{24}$ 

	80		CHAPTER 4. DATATYPES		
1 2		CONTIGUOUS(COUNT, OLDTYPE ER COUNT, OLDTYPE, NEWTYP			
3 4 5		newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using <i>extent</i> as the size of the concatenated copies.			
6 7 8	-	<b>4.2</b> Let oldtype have type m The type map of the datatype	ap $\{(double, 0), (char, 8)\}$ , with extent 16, and let e returned by <b>newtype</b> is		
9	{(doi	uble,0),(char,8),(double,16),(double,	$char, 24), (double, 32), (char, 40)\};$		
10 11	i.e., alterna	ating double and char elements	s, with displacements $0, 8, 16, 24, 32, 40$ .		
12 13 14	In gen	eral, assume that the type ma	p of <b>oldtype</b> is		
15	$\{(typ)\}$	$(type_0, disp_0), \dots, (type_{n-1}, disp_{n-2})$	1)},		
16 17	with exten	t $ex$ . Then newtype has a type	e map with $count \cdot n$ entries defined by:		
18	$\{(typ)\}$	$(be_0, disp_0), \dots, (type_{n-1}, disp_{n-2})$	$(type_0, disp_0 + ex),, (type_{n-1}, disp_{n-1} + ex),$		
19 20	( <i>t</i>	$upe_0, disp_0 + ex \cdot (count - 1)).$	$\dots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$		
21 22	, (0,	<i>gp</i> :0,, <i>p</i> ( + (	$(\mathbb{C}_{\mathcal{F}}^{\mathcal{C}})^{-1},\mathbb{C}_{\mathcal{F}}^{\mathcal{C}})^{-1} = \mathbb{C}_{\mathcal{F}}^{\mathcal{C}} (\mathbb{C}_{\mathcal{F}}^{\mathcal{C}})^{-1} = \mathbb{C}_{\mathcal{F}}^{\mathcal{C}} (\mathbb{C})^{-1} = \mathbb{C}_{\mathcal{F}}^{\mathcal{C}} ($		
23 24 25 26 27 28 29	Vector The function MPI_TYPE_VECTOR is a more general constructor that all cation of a datatype into locations that consist of equally spaced blocks. Each obtained by concatenating the same number of copies of the old datatype. The between blocks is a multiple of the extent of the old datatype.				
30	MPI_TYPE	E_VECTOR(count, blocklength,	stride, oldtype, newtype)		
31 32	IN	count	number of blocks (non-negative integer)		
33 34	IN	blocklength	number of elements in each block (non-negative integer)		
35 36	IN	stride	number of elements between start of each block (integer)		
37 38	IN	oldtype	old datatype (handle)		
39	OUT	newtype	new datatype (handle)		
40 41 42	int MPI_T		blocklength, int stride, , MPI_Datatype *newtype)		
43 44 45 46 47 48	<pre>MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)     BIND(C)     INTEGER, INTENT(IN) :: count, blocklength, stride     TYPE(MPI_Datatype), INTENT(IN) :: oldtype     TYPE(MPI_Datatype), INTENT(OUT) :: newtype</pre>				
-					

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)	2 3
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	4
	5
<b>Example 4.3</b> 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	6 7
type map,	8
$\{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40), \}$	9 10
	10
$({\sf double}, 64), ({\sf char}, 72), ({\sf double}, 80), ({\sf char}, 88), ({\sf double}, 96), ({\sf char}, 104) \}.$	12
That is, two blocks with three copies each of the old type, with a stride of 4 elements (4 $\cdot16$	13 14
bytes) between the the start of each block.	15
<b>Example 4.4</b> A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the	16
datatype,	17 18
$\{(double,0),(char,8),(double,-32),(char,-24),(double,-64),(char,-56)\}.$	19
	20
In general, assume that oldtype has type map,	21 22
$\{(type_0, disp_0),, (type_{n-1}, disp_{n-1})\},\$	23
	24 25
with extent $ex$ . Let bl be the blocklength. The newly created datatype has a type map with count $\cdot$ bl $\cdot n$ entries:	25 26 27
$\{(type_0, disp_0),, (type_{n-1}, disp_{n-1}),$	28
$(type_0, disp_0 + ex),, (type_{n-1}, disp_{n-1} + ex),,$	29 30
	31
$(type_0, disp_0 + (bl - 1) \cdot ex),, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$	32
$(type_0, disp_0 + stride \cdot ex),, (type_{n-1}, disp_{n-1} + stride \cdot ex),,$	33 34
$(type_0, disp_0 + (stride + bl - 1) \cdot ex),, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex),,$	35
	36
$(type_0, disp_0 + stride \cdot (count - 1) \cdot ex),,$	37 38
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex),,$	39
$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex),,$	40 41
	42
$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$	43
	44 45
A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to	46
MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1,	47
count, n, oldtype, newtype), n arbitrary.	48

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

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

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)

	newtype)		15
IN	count	number of blocks – also number of entries in	16
		$array_of_displacements and array_of_blocklengths (non-$	17
		negative integer)	18
IN	array_of_blocklengths	number of elements per block (array of non-negative	19
		integers)	20
IN	array_of_displacements	displacement for each block, in multiples of oldtype	21 22
		extent (array of integer)	22
IN	oldtype	old datatype (handle)	24
OUT	newtype	new datatype (handle)	25
001	newcype	new dubutype (numate)	26

### 

INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(\*), ARRAY\_OF\_DISPLACEMENTS(\*),
OLDTYPE, NEWTYPE, IERROR

### 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,

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

#### Unofficial Draft for Comment Only

```
1
              (\mathsf{double}, 0), (\mathsf{char}, 8)\}.
\mathbf{2}
3
       That is, three copies of the old type starting at displacement 64, and one copy starting at
4
       displacement 0.
5
6
             In general, assume that oldtype has type map,
7
8
              \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\
9
                                 Let B be the array_of_blocklengths argument and D be the
       with extent ex.
10
       array_of_displacements argument. The newly created datatype has n \cdot \sum_{i=0}^{\text{count}-1} B[i] entries:
11
12
              \{(type_0, disp_0 + D[0] \cdot ex), ..., (type_{n-1}, disp_{n-1} + D[0] \cdot ex), ..., \}
13
              (type_0, disp_0 + (D[0] + B[0] - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), ...,
14
15
16
              (type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] \cdot ex), ...,
17
              (type_0, disp_0 + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,
18
19
              (type_{n-1}, disp_{n-1} + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.
20
21
22
23
             A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent
^{24}
       to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where
25
              \mathsf{D}[\mathsf{j}] = j \cdot \mathsf{stride}, \ j = 0, \dots, \mathsf{count} - 1,
26
27
       and
28
29
              B[j] = blocklength, j = 0, ..., count - 1.
30
^{31}
       Hindexed The function MPI_TYPE_CREATE_HINDEXED is identical to
32
       MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are spec-
33
       ified in bytes, rather than in multiples of the oldtype extent.
34
35
       MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements,
36
37
                          oldtype, newtype)
38
         IN
                      count
                                                          number of blocks — also number of entries in
39
                                                          array_of_displacements and array_of_blocklengths (non-
40
                                                          negative integer)
41
         IN
                      array_of_blocklengths
                                                          number of elements in each block (array of non-negative
42
                                                          integers)
43
44
         IN
                      array_of_displacements
                                                          byte displacement of each block (array of integer)
45
         IN
                      oldtype
                                                          old datatype (handle)
46
         OUT
                                                          new datatype (handle)
                      newtype
47
48
```

1 int MPI\_Type\_create\_hindexed(int count, const int array\_of\_blocklengths[], 2 const MPI\_Aint array\_of\_displacements[], MPI\_Datatype oldtype, 3 MPI\_Datatype \*newtype) 4 MPI\_Type\_create\_hindexed(count, array\_of\_blocklengths, 5array\_of\_displacements, oldtype, newtype, ierror) 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) :: oldtype 10 TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI\_TYPE\_CREATE\_HINDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS, 14ARRAY\_OF\_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) 15INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(\*), OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) ARRAY\_OF\_DISPLACEMENTS(\*) 161718 19 Assume that oldtype has type map, 20 $\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},\$ 2122 with extent *ex*. Let B be the array\_of\_blocklengths argument and D be the 23array\_of\_displacements argument. The newly created datatype has a type map with  $n \cdot$ 24 $\sum_{i=0}^{\text{count}-1} B[i]$  entries: 25 $\{(type_0, disp_0 + D[0]), ..., (type_{n-1}, disp_{n-1} + D[0]), ..., \}$ 2627 $(type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex), ...,$ 28 29  $(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), ...,$ 30 31  $(type_0, disp_0 + D[count-1]), ..., (type_{n-1}, disp_{n-1} + D[count-1]), ...,$ 32 33  $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,$ 34 35 $(type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$ 36 37 38 39

Indexed\_block This function is the same as MPI\_TYPE\_INDEXED except that the blocklength is the same for all blocks. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). The following convenience function allows for constant blocksize and arbitrary displacements.

> 45 46

40

41

42

43 44

1 2	MPI_TYP	E_CREATE_INDEXED_BLOC newtype)	K(count, blocklength, array_of_displacements, oldtype,	
$\frac{3}{4}$	IN	count	length of array of displacements (non-negative integer)	
5	IN	blocklength	size of block (non-negative integer)	
6	IN	array_of_displacements	array of displacements (array of integer)	
7	IN	oldtype	old datatype (handle)	
8 9	OUT	newtype	new datatype (handle)	
10				
11 12 13	int MPI_'	• 1	x(int count, int blocklength, const cements[], MPI_Datatype oldtype, pe)	
14 15 16 17 18 19 20 21	INTE arra TYPE TYPE	_create_indexed_block(cou oldtype, newtype, i GER, INTENT(IN) :: count y_of_displacements(count) (MPI_Datatype), INTENT(IN (MPI_Datatype), INTENT(OU GER, OPTIONAL, INTENT(OU	t, blocklength, ) N) :: oldtype JT) :: newtype	
22 23 24 25 26 27 28 29 30	<ul> <li>MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)</li> <li>INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR</li> <li>Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.</li> </ul>			
31 32 33	MPI_TYP	E_CREATE_HINDEXED_BLO oldtype, newtype)	CK(count, blocklength, array_of_displacements,	
34	IN	count	length of array of displacements (non-negative integer)	
35 36	IN	blocklength	size of block (non-negative integer)	
37	IN	array_of_displacements	byte displacement of each block (array of integer)	
38	IN	oldtype	old datatype (handle)	
39 40	OUT	newtype	new datatype (handle)	
41	001	newtype	new datatype (nandle)	
42 43 44 45	<pre>int MPI_Type_create_hindexed_block(int count, int blocklength, const</pre>			
46 47 48		_create_hindexed_block(co oldtype, newtype, i GER, INTENT(IN) :: count		

	EGER(KIND=MPI_ADDRESS_KIN		1 2
	ay_of_displacements(count E(MPI_Datatype), INTENT(I		3
	E(MPI_Datatype), INTENT(I E(MPI_Datatype), INTENT(O		4
	EGER, OPTIONAL, INTENT(OU	· · ·	5
			6
MPI_TYP		OUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	7
T. 1 (7)	OLDTYPE, NEWTYPE, J		8
	EGER COUNT, BLOCKLENGTH,		9
	EGER(KIND=MPI_ADDRESS_KIN	D) ARRAY_OF_DISPLACEMENTS(*)	10
			11
Struct I	MPI_TYPE_CREATE_STRUC	<b>F</b> is the most general type constructor. It further	12
generaliz	es MPI_TYPE_CREATE_HIND	DEXED in that it allows each block to consist of repli-	13
cations o	f different datatypes.		14
			15
	DE CREATE STRUCT(count	array of blocklangths array of displacements	16 17
	array_of_types, newtyp	array_of_blocklengths, array_of_displacements, e)	18
IN	count	number of blocks (non-negative integer) — also num-	19
		ber of entries in arrays array_of_types,	20
		$array_of_displacements and array_of_blocklengths$	21 22
IN	array_of_blocklength	number of elements in each block (array of non-negative	22
	, 3	integer)	23 24
IN	array_of_displacements	byte displacement of each block (array of integer)	25
IN	array_of_types	type of elements in each block (array of handles to datatype objects)	26 27
0.UT		° <u>−</u> ° ,	28
OUT	newtype	new datatype (handle)	29
int MPT	Type create struct(int c	<pre>ount, const int array_of_blocklengths[],</pre>	30 31
1110 111 1.		<pre>y_of_displacements[], const</pre>	32
		of_types[], MPI_Datatype *newtype)	33
			34
MPI_Type	e_create_struct(count, ar		35
	· 1	ents, array_of_types, newtype, ierror)	36
ד אוסיו	BIND(C)	++ +1+1++ - (+ )	37
		t, array_of_blocklengths(count)	38
	EGER(KIND=MPI_ADDRESS_KIN ay_of_displacements(count		39
		N) :: array_of_types(count)	40
	E(MPI_Datatype), INTENT(D		41
	EGER, OPTIONAL, INTENT(OU	• -	42
			43
MPI_TYP	E_CREATE_STRUCT(COUNT, AR		44
- 1. m		ENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)	45
		KLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,	46
IERI			47
TNT	EGEV (VIND-LILI ADDKE99 VIN	<pre>D) ARRAY_OF_DISPLACEMENTS(*)</pre>	48

1	Example 4.6 Let type1 have type map,
2 3	$\{(double, 0), (char, 8)\},\$
4 5 6 7	with extent 16. Let $B = (2, 1, 3)$ , $D = (0, 16, 26)$ , and $T = (MPI_FLOAT, type1, MPI_CHAR)$ . Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,
8	$\{({\sf float},0),({\sf float},4),({\sf double},16),({\sf char},24),({\sf char},26),({\sf char},27),({\sf char},28)\}.$
9 10 11 12	That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)
13 14	In general, let T be the array_of_types argument, where T[i] is a handle to,
15	
16 17	$typemap_{i} = \{(type_{0}^{i}, disp_{0}^{i}),, (type_{n_{i}-1}^{i}, disp_{n_{i}-1}^{i})\},\$
18 19 20	with extent $ex_i$ . Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{c-1} B[i] \cdot n_i$ entries:
21	$\{(type_{0}^{0}, disp_{0}^{0} + D[0]),, (type_{n_{0}}^{0}, disp_{n_{0}}^{0} + D[0]),,$
22 23 24	$(type_{0}^{0}, disp_{0}^{0} + D[0] + (B[0] - 1) \cdot ex_{0}),, (type_{n_{0}}^{0}, disp_{n_{0}}^{0} + D[0] + (B[0] - 1) \cdot ex_{0}),,$
25 26	$(type_{0}^{c-1}, disp_{0}^{c-1} + D[c\text{-1}]),, (type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c\text{-1}]),,$
27	$(type_0^{c-1}, disp_0^{c-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{c-1}),,$
28 29	$(type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1] + (B[c-1]-1) \cdot ex_{c-1})\}.$
$30 \\ 31$	
32 33	A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T
34	is equal to oldtype.
35 36	
37	
38	
39 40	
40 41	
42	
43	
44	
45	
46 47	
48	

#### 4.1. DERIVED DATATYPES

#### 4.1.3 Subarray Datatype Constructor

# MPI\_TYPE\_CREATE\_SUBARRAY(ndims, array\_of\_sizes, array\_of\_subsizes, array\_of\_starts, order, oldtype, newtype)

			6
IN	ndims	number of array dimensions (positive integer)	7
IN	array_of_sizes	number of elements of type $oldtype$ in each dimension	8
		of the full array (array of positive integers)	9
IN	array_of_subsizes	number of elements of type oldtype in each dimension	10
		of the subarray (array of positive integers)	11
			12
IN	array_of_starts	starting coordinates of the subarray in each dimension	13
		(array of non-negative integers)	14
IN	order	array storage order flag (state)	15
IN	oldtype	array element datatype (handle)	16
	olatype		17
OUT	newtype	new datatype (handle)	18

ARRAY\_OF\_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY\_OF\_SIZES(\*), ARRAY\_OF\_SUBSIZES(\*), ARRAY\_OF\_STARTS(\*), ORDER, OLDTYPE, NEWTYPE, IERROR

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 491.

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, re-

 $^{24}$ 

1	spectively. For any dimension i, it is erroneous to specify $array_of_subsizes[i] < 1$ or
2 3	array_of_subsizes[i] > array_of_sizes[i]. The array_of_starts contains the starting coordinates of each dimension of the subarray.
4	Arrays are assumed to be indexed starting from zero. For any dimension $i$ , it is erroneous to
5	specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).
6	specify analytic statisfies $\langle 0 \rangle$ of analytic statisfies $\langle 0 \rangle$ analytic statisfies $\langle 0 \rangle$ .
7	Advice to users. In a Fortran program with arrays indexed starting from 1, if the
8	starting coordinate of a particular dimension of the subarray is $n,$ then the entry in
9	array_of_starts for that dimension is n-1. (End of advice to users.)
10	
11 12	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:
13 14	$\ensuremath{MPI_ORDER_C}$ The ordering used by C arrays, (i.e., row-major order)
15 16	<b>MPI_ORDER_FORTRAN</b> The ordering used by Fortran arrays, (i.e., column-major order)
17	A ndims-dimensional subarray (newtype) with no extra padding can be defined by the
18	function Subarray() as follows:
19	newtype = Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },
20	{ $subsize_0, subsize_1, \dots, subsize_{ndims-1}$ },
21	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$
22 23	$\{start_0, start_1, \ldots, start_{ndims-1}\}, outype\}$
24	Let the typemap of <b>oldtype</b> have the form:
25 26	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$
27	where $type_i$ is a predefined MPI datatype, and let $ex$ be the extent of oldtype. Then we define
28	the Subarray() function recursively using the following three equations. Equation $4.2$ defines
29	the base step. Equation 4.3 defines the recursion step when $order = MPI_ORDER_FORTRAN$ ,
30	and Equation 4.4 defines the recursion step when $order = MPI_ORDER_C$ . These equations
31	use the conceptual datatypes lb_marker and ub_marker, see Section 4.1.6 on page 104 for
32	details.
33 34	
35	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, $ $(4.2)$
36	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$
37	$= \{(lb_marker, 0),$
38	$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$
39	
40	$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$
41	$disp_{n-1} + (start_0 + 1) \times ex), \dots$
42	$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots,$
43	$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$
44 45	$(ub\_marker, size_0 \times ex) \}$
40	
47	$Subarray(ndims, \{size_0, size_1, \dots, size_{ndims-1}\}, $ $(4.3)$
48	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$
	(·····································

$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	
= Subarray $(ndims - 1, \{size_1, size_2, \dots, size_{ndims - 1}\},\$	
$\{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\},\$	
$\{start_1, start_2, \ldots, start_{ndims-1}\},\$	
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$	
Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },	(4.4)
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$	1
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	:
= Subarray( $ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$	:
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\},\$	:
$\{start_0, start_1, \ldots, start_{ndims-2}\},\$	-
Subarray(1, $\{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldt$	ype))

For an example use of MPI\_TYPE\_CREATE\_SUBARRAY in the context of I/O see Section 13.9.2.

#### 4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [43] 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 on page 491 and Section 13.3 on page 503. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

 $\frac{24}{25}$ 

```
1
     MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs,
\mathbf{2}
                     array_of_dargs, array_of_psizes, order, oldtype, newtype)
3
       IN
                 size
                                              size of process group (positive integer)
4
       IN
                                              rank in process group (non-negative integer)
                 rank
5
6
       IN
                 ndims
                                              number of array dimensions as well as process grid
7
                                              dimensions (positive integer)
8
       IN
                 array_of_gsizes
                                              number of elements of type oldtype in each dimension
9
                                              of global array (array of positive integers)
10
                 array_of_distribs
       IN
                                              distribution of array in each dimension (array of state)
11
12
       IN
                 array_of_dargs
                                              distribution argument in each dimension (array of pos-
13
                                              itive integers)
14
       IN
                 array_of_psizes
                                              size of process grid in each dimension (array of positive
15
                                              integers)
16
       IN
                 order
                                              array storage order flag (state)
17
18
       IN
                 oldtype
                                              old datatype (handle)
19
       OUT
                                              new datatype (handle)
                 newtype
20
21
     int MPI_Type_create_darray(int size, int rank, int ndims, const
22
                     int array_of_gsizes[], const int array_of_distribs[], const
23
                     int array_of_dargs[], const int array_of_psizes[], int order,
^{24}
                     MPI_Datatype oldtype, MPI_Datatype *newtype)
25
26
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
27
                     array_of_distribs, array_of_dargs, array_of_psizes, order,
28
                     oldtype, newtype, ierror) BIND(C)
29
          INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
30
          array_of_distribs(ndims), array_of_dargs(ndims),
31
          array_of_psizes(ndims), order
32
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                   oldtype
33
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
34
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
36
                     ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
37
                     OLDTYPE, NEWTYPE, IERROR)
38
          INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
39
          ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
40
41
          MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to
42
     the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional
43
     grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See
44
     Example 4.7, page 101.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the
     equation \prod_{i=0}^{ndims-1} array_of_psizes[i] = size must be satisfied. The ordering of processes
45
46
     in the process grid is assumed to be row-major, as in the case of virtual Cartesian process
47
     topologies.
48
```

CHAPTER 4. DATATYPES

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7 on page 289. (End of advice to users.) Each dimension of the array can be distributed in one of three ways: • MPI\_DISTRIBUTE\_BLOCK - Block distribution

- MPI\_DISTRIBUTE\_CYCLIC Cyclic distribution
- MPI\_DISTRIBUTE\_NONE Dimension not distributed.

The constant MPI\_DISTRIBUTE\_DFLT\_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI\_DISTRIBUTE\_BLOCK, it is erroneous to specify array\_of\_dargs[i] \* array\_of\_psizes[i] < array\_of\_gsizes[i].

For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to MPI\_DISTRIBUTE\_CYCLIC with a distribution argument of 15, and the HPF layout AR-RAY(BLOCK) corresponds to MPI\_DISTRIBUTE\_BLOCK with a distribution argument of MPI\_DISTRIBUTE\_DFLT\_DARG.

The order argument is used as in MPI\_TYPE\_CREATE\_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI\_ORDER\_FORTRAN and MPI\_ORDER\_C.

This routine creates a new MPI datatype with a typemap defined in terms of a function called "cyclic()" (see below).

```
Without loss of generality, it suffices to define the typemap for the
```

MPI\_DISTRIBUTE\_CYCLIC case where MPI\_DISTRIBUTE\_DFLT\_DARG is not used. MPI\_DISTRIBUTE\_BLOCK and MPI\_DISTRIBUTE\_NONE can be reduced to the MPI\_DISTRIBUTE\_CYCLIC case for dimension i as follows.

```
MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG
is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to
```

 $(array_of_gsizes[i] + array_of_psizes[i] - 1)/array_of_psizes[i].$ 

If array\_of\_dargs[i] is not MPI\_DISTRIBUTE\_DFLT\_DARG, then MPI\_DISTRIBUTE\_BLOCK and 37 MPI\_DISTRIBUTE\_CYCLIC are equivalent. MPI\_DISTRIBUTE\_NONE is equivalent to MPI\_DISTRIBUTE\_CYCLIC with

array\_of\_dargs[i] set to array\_of\_gsizes[i].

Finally, MPI\_DISTRIBUTE\_CYCLIC with array\_of\_dargs[i] equal to MPI\_DISTRIBUTE\_DFLT\_DARG is equivalent to MPI\_DISTRIBUTE\_CYCLIC with 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; for ( i = 0; i < ndims; i++ ) {</pre> 1

 $\mathbf{2}$ 

3

4

56

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

3536

38

39

40

41

42

43

44

4546

47

```
1
              oldtypes[i+1] = cyclic(array_of_dargs[i],
2
                                           array_of_gsizes[i],
3
                                           r[i],
4
                                           array_of_psizes[i],
5
                                           oldtypes[i]);
6
           }
7
           newtype = oldtypes[ndims];
8
          For MPI_ORDER_C, the code is:
9
10
           oldtypes[0] = oldtype;
11
           for ( i = 0; i < ndims; i++ ) {</pre>
12
              oldtypes[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
13
                                              array_of_gsizes[ndims - i - 1],
14
                                              r[ndims - i - 1],
15
                                              array_of_psizes[ndims - i - 1],
16
                                              oldtypes[i]);
17
           }
18
           newtype = oldtypes[ndims];
19
20
21
      where r[i] is the position of the process (with rank rank) in the process grid at dimension i.
22
      The values of r[i] are given by the following code fragment:
23
24
                t_rank = rank;
25
                t_size = 1;
26
                for (i = 0; i < ndims; i++)</pre>
27
                          t_size *= array_of_psizes[i];
28
                for (i = 0; i < ndims; i++) {
29
                     t_size = t_size / array_of_psizes[i];
30
                     r[i] = t_rank / t_size;
31
                     t_rank = t_rank % t_size;
32
                }
33
34
           Let the typemap of oldtype have the form:
35
            \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
36
37
      where type_i is a predefined MPI datatype, and let ex be the extent of
38
      oldtype. The following function uses the conceptual datatypes lb_marker and ub_marker, see
39
      Section 4.1.6 on page 104 for details.
40
           Given the above, the function cyclic() is defined as follows:
41
42
            cyclic(darg, gsize, r, psize, oldtype)
43
              = {(lb_marker, 0),
44
                  (type_0, disp_0 + r \times darg \times ex), \ldots,
45
                          (type_{n-1}, disp_{n-1} + r \times darg \times ex),
46
                  (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots,
47
48
                          (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex),
```

	1
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots,$	2
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),$	3
	4 5
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \dots,$	6
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$	7
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,$	8
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$	9
$(g_{p}\circ_{n-1}, a\circ\circ_{p}\circ_{n-1})$ , $(f, f, aa, g + 2)$ , $(a + p\circ n \circ f, aa, g + o \circ f)$ ,	10 11
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$	12
$(type_0, usp_0 + ((r+1) \times usrg - 1) \times ex + psize \times usrg \times ex), \dots, $ $(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),$	13
$(igpe_{n-1}, aisp_{n-1} + ((i+1) \land aaig = 1) \land ex + psize \land aaig \land ex),$	14
:	15
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	16
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$	17 18
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	19
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$	20
+psize  imes darg  imes ex  imes (count - 1)),	21
	22
$(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$	23 24
$+psize  imes darg  imes ex  imes (count - 1)), \dots,$	24 25
$(type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex$	26
$+psize \times darg \times ex \times (count - 1)),$	27
$(ub\_marker, gsize * ex)\}$	28
where <i>count</i> is defined by this code fragment:	29 30
<pre>nblocks = (gsize + (darg - 1)) / darg;</pre>	31
count = nblocks / psize;	32
<pre>left_over = nblocks - count * psize;</pre>	33
if (r < left_over)	34
<pre>count = count + 1;</pre>	35
Here, <i>nblocks</i> is the number of blocks that must be distributed among the processors.	36 37
Finally, $darg_{last}$ is defined by this code fragment:	38
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)	39
<pre>darg_last = darg;</pre>	40
else	41
<pre>darg_last = num_in_last_cyclic - darg * r; if (darg_last &gt; darg)</pre>	42 43
darg_last = darg;	43 44
if (darg_last <= 0)	45
darg_last = darg;	46
	47

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

## **Unofficial Draft for Comment Only**

```
1
            <oldtype> FILEARRAY(100, 200, 300)
\mathbf{2}
     !HPF$ PROCESSORS PROCESSES(2, 3)
3
     !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
4
     This can be achieved by the following Fortran code, assuming there will be six processes
5
     attached to the run:
6
7
          ndims = 3
8
          array_of_gsizes(1) = 100
9
          array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
10
          array_of_dargs(1) = 10
11
          array_of_gsizes(2) = 200
12
          array_of_distribs(2) = MPI_DISTRIBUTE_NONE
13
          \operatorname{array_of_dargs}(2) = 0
14
          array_of_gsizes(3) = 300
15
          array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
16
          array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
17
          array_of_psizes(1) = 2
18
          array_of_psizes(2) = 1
19
          array_of_psizes(3) = 3
20
          call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
21
          call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
22
          call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
23
               array_of_distribs, array_of_dargs, array_of_psizes,
                                                                                  &
24
               MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
25
26
     4.1.5 Address and Size Functions
27
28
     The displacements in a general datatype are relative to some initial buffer address. Abso-
     lute addresses can be substituted for these displacements: we treat them as displacements
29
30
     relative to "address zero," the start of the address space. This initial address zero is indi-
^{31}
     cated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of
32
     the entries in the communication buffer, in which case the buf argument is passed the value
33
     MPI_BOTTOM.
34
          The address of a location in memory can be found by invoking the function
35
     MPI_GET_ADDRESS.
36
37
     MPI_GET_ADDRESS(location, address)
38
39
       IN
                 location
                                            location in caller memory (choice)
40
       OUT
                                            address of location (integer)
                 address
41
42
     int MPI_Get_address(const void *location, MPI_Aint *address)
43
44
     MPI_Get_address(location, address, ierror) BIND(C)
45
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: location
46
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

Returns the (byte) address of location.

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than  $2^{32} - 1$  are used in the program. New codes should be written so that they use the new functions. This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

*Rationale.* In the mpi\_f08 module, the location argument is not defined with INTENT(IN) because existing applications may use MPI\_GET\_ADDRESS (or the deprecated MPI\_ADDRESS) as a substitute for MPI\_F\_SYNC\_REG that was not defined before MPI-3.0. (*End of rationale.*)

```
Example 4.8 Using MPI_GET_ADDRESS for an array.
```

```
REAL A(100,100)
INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of MPI\_GET\_ADDRESS and rely on the availability of the address operator &. Note, however, that & cast-expression is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI\_GET\_ADDRESS to "reference" C variables guarantees portability to such machines as well. (End of advice to users.)

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. In particular, refer to Sections 17.1.12 and 17.1.13 on pages 628-630 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and Sections 17.1.16-17.1.19 on pages 633-643 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*End of advice to users.*)

 $\mathbf{2}$ 

 $^{24}$ 

```
1
         The following auxiliary functions provide useful information on derived datatypes.
\mathbf{2}
3
     MPI_TYPE_SIZE(datatype, size)
4
5
       IN
                datatype
                                            datatype (handle)
6
       OUT
                size
                                            datatype size (integer)
7
8
     int MPI_Type_size(MPI_Datatype datatype, int *size)
9
10
     MPI_Type_size(datatype, size, ierror) BIND(C)
11
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
12
         INTEGER, INTENT(OUT) :: size
13
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
14
     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
15
          INTEGER DATATYPE, SIZE, IERROR
16
17
18
     MPI_TYPE_SIZE_X(datatype, size)
19
20
       IN
                datatype
                                            datatype (handle)
21
       OUT
                size
                                            datatype size (integer)
22
23
     int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
^{24}
25
     MPI_Type_size_x(datatype, size, ierror) BIND(C)
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::
                                                             size
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
30
         INTEGER DATATYPE, IERROR
^{31}
         INTEGER(KIND = MPI_COUNT_KIND) SIZE
32
33
         MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to the total size, in
34
```

<sup>34</sup> bytes, of the entries in the type signature associated with datatype; i.e., the total size, in <sup>35</sup> data in a message that would be created with this datatype. Entries that occur multiple <sup>36</sup> times in the datatype are counted with their multiplicity. For both functions, if the OUT <sup>37</sup> parameter cannot express the value to be returned (e.g., if the parameter is too small to <sup>38</sup> hold the output value), it is set to MPI\_UNDEFINED.

40 41

## 4.1.6 Lower-Bound and Upper-Bound Markers

<sup>42</sup> It is often convenient to define explicitly the lower bound and upper bound of a type map, <sup>43</sup> and override the definition given on page 105. This allows one to define a datatype that has <sup>44</sup> "holes" at its beginning or its end, or a datatype with entries that extend above the upper <sup>45</sup> bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. <sup>46</sup> Also, the user may want to overide the alignment rules that are used to compute upper <sup>47</sup> bounds and extents. E.g., a C compiler may allow the user to overide default alignment <sup>48</sup> rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional conceptual datatypes,  $lb_marker$  and  $ub_marker$ , that represent the lower bound and upper bound of a datatype. These conceptual datatypes occupy no space (*extent*( $lb_marker$ ) = *extent*( $ub_marker$ ) = 0). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

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

In general, if

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

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

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

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

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

Then

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

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

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

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

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

22

25

26

30 31

32 33

34

35

36

37

38

39

40

 $41 \\ 42$ 

43

44

45

46

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 PRECISION data. Such gaps may be added explicitly to both the structure and the MPI derived datatype handle because the communication of a contiguous derived datatype 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 DOUB	LE PRECISION is
! two times the size of a RE	AL
DOUBLE PRECISION:: p	
END TYPE	
one should define	
TYPE, BIND(C) :: my_data	
, , , <u>, , , , , , , , , , , , , , , , </u>	

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 631. (*End of advice to users.*)

1

2

3

4

5

7

8

9

10

11

12

13

14

15

16 17

 $^{31}$ 

32

33

34 35 36

37

38

39

40

41

42

43

44

45

4.1.7 E	xtent and Bounds of	Datatypes	1
		51	2
			3
MPI_TYPE_GET_EXTENT(datatype, lb, extent)			4
IN	datatype	datatype to get information on (handle)	5
			6
OUT	lb	lower bound of datatype (integer)	7 8
OUT	extent	extent of datatype (integer)	9
			10
int MPI		PI_Datatype datatype, MPI_Aint *1b,	11
	MPI_Aint *ex	ctent)	12
MPI_Type	e_get_extent(datat	<pre>ype, lb, extent, ierror) BIND(C)</pre>	13
	-	NTENT(IN) :: datatype	14
INT	EGER(KIND=MPI_ADDR	ESS_KIND), INTENT(OUT) :: lb, extent	15
INT	EGER, OPTIONAL, IN	TENT(OUT) :: ierror	16
MPI_TYP	E_GET_EXTENT(DATAT	YPE, LB, EXTENT, IERROR)	17
	EGER DATATYPE, IER		18 19
INT	EGER(KIND = MPI_AD	DRESS_KIND) LB, EXTENT	20
			21
			22
MPI_TYI	PE_GET_EXTENT_X	(datatype, lb, extent)	23
IN	datatype	datatype to get information on (handle)	24
OUT	lb	lower bound of datatype (integer)	25
OUT	extent		26
001	extent	extent of datatype (integer)	27
int MDT	Turno got outont w	(MPI_Datatype datatype, MPI_Count *1b,	28 29
IIIC MFI.	MPI_Count *e		30
			31
	-	atype, lb, extent, ierror) BIND(C)	32
		NTENT(IN) :: datatype	33
<pre>INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: lb, extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			34
T 11 T 1	GER, OFIIONAL, IN	TENI(UUI) TETIUI	35
		ATYPE, LB, EXTENT, IERROR)	36
	EGER DATATYPE, IER		37
TNT]	GER(KIND = MP1_CO	UNT_KIND) LB, EXTENT	38 39
Retu	irns the lower bound	and the extent of datatype (as defined in Section $4.1.6$ on	40

Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on page 104).

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

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers. This provides control over the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the **count** argument in a send or receive call.

 $^{41}$ 

 $<sup>46 \\ 47</sup>$ 

1 MPI\_TYPE\_CREATE\_RESIZED(oldtype, lb, extent, newtype) 2 IN oldtype input datatype (handle) 3 IN lb new lower bound of datatype (integer) 4 5IN new extent of datatype (integer) extent 6 OUT newtype output datatype (handle) 7 8 int MPI\_Type\_create\_resized(MPI\_Datatype oldtype, MPI\_Aint lb, MPI\_Aint 9 extent, MPI\_Datatype \*newtype) 10 11MPI\_Type\_create\_resized(oldtype, lb, extent, newtype, ierror) BIND(C) 12INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: lb, extent 13 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 14TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI\_TYPE\_CREATE\_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR) 17INTEGER OLDTYPE, NEWTYPE, IERROR 18 INTEGER(KIND=MPI\_ADDRESS\_KIND) LB, EXTENT 19 20Returns in newtype a handle to a new datatype that is identical to oldtype, except that 21the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb 22+ extent. Any previous lb and ub markers are erased, and a new pair of lower bound and 23upper bound markers are put in the positions indicated by the lb and extent arguments.  $^{24}$ This affects the behavior of the datatype when used in communication operations, with 25count > 1, and when used in the construction of new derived datatypes. 2627True Extent of Datatypes 4.1.8 28Suppose we implement gather (see also Section 5.5 on page 149) as a spanning tree im-29 plemented on top of point-to-point routines. Since the receive buffer is only valid on the 30 root process, one will need to allocate some temporary space for receiving data on in- $^{31}$ termediate nodes. However, the datatype extent cannot be used as an estimate of the 32 amount of space that needs to be allocated, if the user has modified the extent, for example 33 by using MPI\_TYPE\_CREATE\_RESIZED. The functions MPI\_TYPE\_GET\_TRUE\_EXTENT 34and MPI\_TYPE\_GET\_TRUE\_EXTENT\_X are provided which return the true extent of the 35 datatype. 36 37 38 MPI\_TYPE\_GET\_TRUE\_EXTENT(datatype, true\_lb, true\_extent) 39 IN datatype datatype to get information on (handle) 4041 OUT true\_lb true lower bound of datatype (integer) 42OUT true\_extent true size of datatype (integer) 43 44int MPI\_Type\_get\_true\_extent(MPI\_Datatype datatype, MPI\_Aint \*true\_lb, 45MPI\_Aint \*true\_extent) 4647MPI\_Type\_get\_true\_extent(datatype, true\_lb, true\_extent, ierror) BIND(C) 48

INTEC	(MPI_Datatype), INTENT(IN GER(KIND=MPI_ADDRESS_KIND GER, OPTIONAL, INTENT(OUT	), INTENT(OUT) :: true_lb, true_extent	1 2 3
INTEC	_GET_TRUE_EXTENT(DATATYPE GER DATATYPE, IERROR GER(KIND = MPI_ADDRESS_KI	, TRUE_LB, TRUE_EXTENT, IERROR) ND) TRUE_LB, TRUE_EXTENT	4 5 6 7 8
MPI_TYPE_GET_TRUE_EXTENT_X(datatype, true_lb, true_extent)			9 10
IN	datatype	datatype to get information on (handle)	11
OUT	true_lb	true lower bound of datatype (integer)	12 13
OUT	true_extent	true size of datatype (integer)	14
int MPI_7	Type_get_true_extent_x(MP MPI_Count *true_exte	I_Datatype datatype, MPI_Count *true_lb, ent)	15 16 17
<pre>MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror) BIND(C)     TYPE(MPI_Datatype), INTENT(IN) :: datatype     INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			18 19 20 21 22
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND = MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT			23 24 25
true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring explicit lower bound and upper bound markers, and performing no rounding for alignment. If the typemap associated with datatype is			26 27 28 29 30 31
Type	$emap = \{(type_0, disp_0), \dots, (t_n)\}$	$ype_{n-1}, disp_{n-1})\}$	32
Then			33 34
$true_{i}$	$lb(Typemap) = min_j \{ disp_j \}$	: $type_j \neq lb_marker, ub_marker\},$	35 36
$true_{i}$	$\_ub(Typemap) = max_j \{ disp_j \}$	$+ sizeof(type_j) : type_j \neq lb_marker, ub_marker\},$	37 38
and			39
true	$\_extent(Typemap) = true\_ub$	$(Typemap) - true\_lb(typemap).$	40 41
tion 4.1.7 The t datatype,	on page 107, which describe t true_extent is the minimum uncompressed.	definitions in Section 4.1.6 on page 104 and Seche function MPI_TYPE_GET_EXTENT.) number of bytes of memory necessary to hold a	42 43 44 45 46 47
(e.g., if the	e parameter is too small to he	old the output value), it is set to MPI_UNDEFINED.	48

```
\mathbf{2}
     A datatype object has to be committed before it can be used in a communication. As
3
     an argument in datatype constructors, uncommitted and also committed datatypes can be
4
     used. There is no need to commit basic datatypes. They are "pre-committed."
5
6
7
     MPI_TYPE_COMMIT(datatype)
8
       INOUT
                 datatype
                                             datatype that is committed (handle)
9
10
     int MPI_Type_commit(MPI_Datatype *datatype)
11
12
     MPI_Type_commit(datatype, ierror) BIND(C)
13
          TYPE(MPI_Datatype), INTENT(INOUT) ::
                                                     datatype
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_TYPE_COMMIT(DATATYPE, IERROR)
16
          INTEGER DATATYPE, IERROR
17
18
         The commit operation commits the datatype, that is, the formal description of a com-
19
     munication buffer, not the content of that buffer. Thus, after a datatype has been commit-
20
     ted, it can be repeatedly reused to communicate the changing content of a buffer or, indeed,
21
     the content of different buffers, with different starting addresses.
22
23
                                     The system may "compile" at commit time an internal
           Advice to implementors.
24
           representation for the datatype that facilitates communication, e.g. change from a
25
           compacted representation to a flat representation of the datatype, and select the most
26
           convenient transfer mechanism. (End of advice to implementors.)
27
28
          MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent
29
     to a no-op.
30
^{31}
     Example 4.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.
32
     INTEGER type1, type2
33
     CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
34
                     ! new type object created
35
     CALL MPI_TYPE_COMMIT(type1, ierr)
36
                     ! now type1 can be used for communication
37
     type2 = type1
38
                      ! type2 can be used for communication
39
                      ! (it is a handle to same object as type1)
40
     CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
41
                     ! new uncommitted type object created
42
     CALL MPI_TYPE_COMMIT(type1, ierr)
43
                      ! now type1 can be used anew for communication
44
45
46
47
48
```

4.1.9

Commit and Free

	MPI_TYPE	E_FREE(datatype)		1
	INOUT	datatypo	deteture that is freed (handle)	2
	INCOT	datatype	datatype that is freed (handle)	3
				4
	int MPI_T	<pre>Sype_free(MPI_Datatype *d</pre>	latatype)	5
	MPI_Type_	_free(datatype, ierror) E	BIND(C)	6
	• 1	MPI_Datatype), INTENT(IN		7
		ER, OPTIONAL, INTENT(OUT		8
				9
		FREE(DATATYPE, IERROR)		10
	INTEG	ER DATATYPE, IERROR		11
	Marks	the datatype object associat	ted with datatype for deallocation and sets datatype	12
		0 × 0	nication that is currently using this datatype will	13
complete normally. Freeing a datatype does not affect any other datatype that was built			14	
	from the freed datatype. The system behaves as if input datatype arguments to derived			15
datatype constructors are passed by value.			16	
aarot	51			17
	Advie	ce to implementors. The im	plementation may keep a reference count of active	18
	comn	nunications that use the data	atype, in order to decide when to free it. Also, one	19

may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (End of advice to implementors.)

#### 4.1.10 Duplicating a Datatype

MPI\_TYPE\_DUP(oldtype, newtype)

IN	oldtype	datatype (handle)
OUT	newtype	copy of <b>oldtype</b> (handle)

int MPI\_Type\_dup(MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

MPI\_Type\_dup(oldtype, newtype, ierror) BIND(C) TYPE(MPI\_Datatype), INTENT(IN) :: oldtype TYPE(MPI\_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_TYPE\_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR

MPI\_TYPE\_DUP is a type constructor which duplicates the existing 42type 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 44particular action that a copy callback may take is to delete the attribute from the new 45datatype. Returns in newtype a new datatype with exactly the same properties as oldtype 46and any copied cached information, see Section 6.7.4 on page 276. The new datatype has 47identical upper bound and lower bound and yields the same net result when fully decoded 48

#### **Unofficial Draft for Comment Only**

20

21

22

23

 $^{24}$ 2526

> 2728

34

35

36

37

38 39

40

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

<sup>4</sup> 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,

```
<sup>11</sup> MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
```

- <sup>12</sup> MPI\_TYPE\_COMMIT(newtype)
- <sup>13</sup> MPI\_SEND(buf, 1, newtype, dest, tag, comm).

<sup>15</sup> Similar statements apply to all other communication functions that have a count and
 <sup>16</sup> datatype argument.

<sup>17</sup> Suppose that a send operation MPI\_SEND(buf, count, datatype, dest, tag, comm) is <sup>18</sup> executed, where datatype has type map,

19 20

30

 $^{31}$ 

32 33  $\{(type_0, disp_0), ..., (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, ..., count - 1 and j = 0, ..., 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), ..., (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 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.

46

- 47
- 48

•••			1
	TYPE_CONTIGUOUS( 2, MPI_F	• =	2 3
	TYPE_CONTIGUOUS( 4, MPI_F		4
	TYPE_CONTIGUOUS( 2, type2	2, type22,)	5
CALL MDT	SEND( a, 4, MPI_REAL,		6
	SEND( a, 4, HII_NEAL,)	.)	7
	SEND( a, 1, type22,)		8
	SEND( a, 1, type4,)		9
			10
CALL MPI_	RECV( a, 4, MPI_REAL,	.)	11
CALL MPI_	RECV( a, 2, type2,)		12
CALL MPI_	RECV( a, 1, type22,)		13
CALL MPI_	RECV( a, 1, type4,)		14
Fach of th	e sends matches any of the re-		15
	0	g entries. The use of such a datatype in a receive	16
		s even if the actual message received is short enough	17
-	e any entry more than once.)	seven if the actual message received is short chough	18 19
	· · · · · · · · · · · · · · · · · · ·	t, datatype, dest, tag, comm, status) is executed,	20
	type has type map,		20
¢ / .			22
$\{(typ)$	$e_0, disp_0), \dots, (type_{n-1}, disp_{n-1})$	1)}.	23
The receive	ed message need not fill all the	e receive buffer, nor does it need to fill a number of	24
	_	number, $k$ , of basic elements can be received, where	25
		lements received can be retrieved from status using	26
the query f	unctions MPI_GET_ELEMEN	TS or MPI_GET_ELEMENTS_X.	27
			28
	ELEMENTS (status datatura	count)	29
MPI_GET_	ELEMENTS(status, datatype,	count)	30
IN	status	return status of receive operation (Status)	31
IN	datatype	datatype used by receive operation (handle)	32 33
OUT	count	number of received basic elements (integer)	34
			35
int MPT G	et elements(const MPI Sta	atus *status, MPI_Datatype datatype,	36
	int *count)		37
NDT G			38
	lements(status, datatype, MPI_Status), INTENT(IN) ;	-	39
	MPI_Status), INTENT(IN) MPI_Datatype), INTENT(IN)		40
	ER, INTENT(OUT) :: count		41
	ER, OPTIONAL, INTENT(OUT)		42
	4		
III 1_delelemento(birito), brixinie, obowi, iendot/			44
INTEG	ER STATUS (MPI_STATUS_SIZE	E), DATATYPE, COUNT, IERROR	45 46
			40
			48

```
1
     MPI_GET_ELEMENTS_X(status, datatype, count)
2
       IN
                 status
                                            return status of receive operation (Status)
3
       IN
                 datatype
                                            datatype used by receive operation (handle)
4
5
       OUT
                                            number of received basic elements (integer)
                 count
6
\overline{7}
     int MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,
8
                    MPI_Count *count)
9
     MPI_Get_elements_x(status, datatype, count, ierror) BIND(C)
10
          TYPE(MPI_Status), INTENT(IN) :: status
11
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
          INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: count
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
16
          INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
17
          INTEGER(KIND=MPI_COUNT_KIND) COUNT
18
         The datatype argument should match the argument provided by the receive call that
19
     set the status variable. For both functions, if the OUT parameter cannot express the value
20
     to be returned (e.g., if the parameter is too small to hold the output value), it is set to
21
     MPI_UNDEFINED.
22
          The previously defined function MPI_GET_COUNT (Section 3.2.5), has a different be-
23
     havior. It returns the number of "top-level entries" received, i.e. the number of "copies" of
^{24}
     type datatype. In the previous example, MPI_GET_COUNT may return any integer value
25
     k, where 0 \le k \le \text{count.} If MPI_GET_COUNT returns k, then the number of basic elements
26
     received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is
27
     n \cdot k. If the number of basic elements received is not a multiple of n, that is, if the receive
28
     operation has not received an integral number of datatype "copies," then MPI_GET_COUNT
29
     sets the value of count to MPI_UNDEFINED.
30
^{31}
     Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.
32
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)
40
            CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
^{41}
     ELSE IF (rank.EQ.1) THEN
42
            CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
43
            CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                               ! returns i=1
44
            CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=2
45
            CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
46
            CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                             ! returns i=MPI_UNDEFINED
47
            CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3
48
     END IF
```

The functions MPI\_GET\_ELEMENTS and MPI\_GET\_ELEMENTS\_X can also be used after a probe to find the number of elements in the probed message. Note that the functions MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X return the same values when they are used with basic datatypes as long as the limits of their respective 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.)

#### 4.1.12 Correct Use of Addresses

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

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

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

A correct program uses only valid addresses to identify the locations of entries in  $^{46}$ communication buffers. Furthermore, if u and v are two valid addresses, then the (integer)  $^{47}$ 

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{31}$ 

<sup>1</sup> difference u - v can be computed only if both u and v are in the same sequential storage. <sup>2</sup> No other arithmetic operations can be meaningfully executed on addresses.

3 The rules above impose no constraints on the use of derived datatypes, as long as 4 they are used to define a communication buffer that is wholly contained within the same  $\mathbf{5}$ sequential storage. However, the construction of a communication buffer that contains 6 variables that are not within the same sequential storage must obey certain restrictions.  $\overline{7}$ Basically, a communication buffer with variables that are not within the same sequential 8 storage can be used only by specifying in the communication call  $buf = MPI_BOTTOM$ , 9 count = 1, and using a datatype argument where all displacements are valid (absolute) 10 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.*)

21 22 23

11

12

13

14

15 16

17

18

19

20

#### 4.1.13 Decoding a Datatype

combiner)

 $^{24}$ MPI datatype objects allow users to specify an arbitrary layout of data in memory. There 25are several cases where accessing the layout information in opaque datatype objects would 26be useful. The opaque datatype object has found a number of uses outside MPI. Further-27more, a number of tools wish to display internal information about a datatype. To achieve 28this, datatype decoding functions are provided. The two functions in this section are used 29 together to decode datatypes to recreate the calling sequence used in their initial defini-30 tion. These can be used to allow a user to determine the type map and type signature of a  $^{31}$ datatype. 32

33 34

```
MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes,
```

		)	
36 37	IN	datatype	datatype to access (handle)
38 39	OUT	num_integers	number of input integers used in the call constructing combiner (non-negative integer)
40 41	OUT	num_addresses	number of input addresses used in the call construct- ing <b>combiner</b> (non-negative integer)
42 43 44	OUT	num_datatypes	number of input datatypes used in the call construct- ing <b>combiner</b> (non-negative integer)
45	OUT	combiner	combiner (state)
46			
47	int MPI_1	<pre>Sype_get_envelope(MPI_Data</pre>	atype datatype, int *num_integers,
48		<pre>int *num_addresses,</pre>	<pre>int *num_datatypes, int *combiner)</pre>

INTEGER DATATYPE, NUM\_INTEGERS, NUM\_ADDRESSES, NUM\_DATATYPES, COMBINER, IERROR

For the given datatype, MPI\_TYPE\_GET\_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI\_TYPE\_GET\_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

*Rationale.* By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

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

The list in Table 4.1 has the values that can be returned in **combiner** on the left and the call associated with them on the right.

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.

1	MDL	COMBINER_NAMED	a named predefined datatype		
2		COMBINER_DUP	MPI_TYPE_DUP		
3		COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS		
	_	COMBINER_VECTOR	MPI_TYPE_VECTOR		
4		COMBINER_HVECTOR			
5	_	—	MPI_TYPE_CREATE_HVECTOR		
6		COMBINER_INDEXED	MPI_TYPE_INDEXED		
7		COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED		
8		COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK		
9		COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK		
10		COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT		
11		COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY		
12		COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY		
13		COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL		
14		COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX		
15	MPI_0	COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER		
16	MPI_0	COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED		
17					
18	т	ble 4.1. combiner values notur	ned from MPI_TYPE_GET_ENVELOPE		
19	1	able 4.1: combiner values retur	nea from MPI_1 TPE_GET_ENVELOPE		
20					
21	MPI_TYPE		max_integers, max_addresses, max_datatypes,		
22		array_of_integers, array_o	f_addresses, array_of_datatypes)		
23	IN	datatype	datatype to access (handle)		
24			• -		
25	IN	max_integers	number of elements in array_of_integers (non-negative		
26			integer)		
27	IN	max_addresses	number of elements in array_of_addresses (non-negative		
28			integer)		
29	IN	max_datatypes	number of elements in array_of_datatypes (non-negative		
30			integer)		
31	0.UT	<b>C</b> • •	<i>o</i> ,		
32	OUT	array_of_integers	contains integer arguments used in constructing		
33			datatype (array of integers)		
34	OUT	array_of_addresses	contains address arguments used in constructing		
35			datatype (array of integers)		
36	OUT	array_of_datatypes	contains datatype arguments used in constructing		
37	001		datatype (array of handles)		
38			datatype (array of handles)		
39	the MDT T				
40	int MPI_1		type datatype, int max_integers,		
41			<pre>t max_datatypes, int array_of_integers[],</pre>		
42		MPI_Aint array_of_add			
43		MPI_Datatype array_of	_datatypes[])		
44	MPI_Tvpe	get_contents(datatvpe. max	x_integers, max_addresses, max_datatypes,		
45	array_of_integers, array_of_addresses, array_of_datatypes,				
46	ierror) BIND(C)				
47	TYPE(MPI_Datatype), INTENT(IN) :: datatype				
48	INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes				
-	TWIEG	, INILAI (IN/ max_III)			

<pre>INTEGER, INTENT(OUT) :: array_of_integers(max_integers)</pre>	1
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::	2
array_of_addresses(max_addresses)	3
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
	6
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	7
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	8
IERROR)	9
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	10
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	11
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)</pre>	12
datatype must be a predefined unnamed or a derived datatype; the call is erroneous if	13
datatype is a predefined named datatype.	14
The values given for max_integers, max_addresses, and max_datatypes must be at least as	15
large as the value returned in num_integers, num_addresses, and num_datatypes, respectively,	16
in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.	17
	18
<i>Rationale.</i> The arguments max_integers, max_addresses, and max_datatypes allow for	19
error checking in the call. ( <i>End of rationale</i> .)	20
The datatypes returned in array_of_datatypes are handles to datatype objects that	21 22
are equivalent to the datatypes used in the original construction call. If these were derived	22
datatypes, then the returned datatypes are new datatype objects, and the user is responsible	23 24
for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then	24 25
the returned datatype is equal to that (constant) predefined datatype and cannot be freed.	26
The committed state of returned derived datatypes is undefined, i.e., the datatypes may	27
or may not be committed. Furthermore, the content of attributes of returned datatypes is	28
undefined.	29
Note that MPI_TYPE_GET_CONTENTS can be invoked with a	30
datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL,	31
MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed	32
predefined datatype). In such a case, an empty array_of_datatypes is returned.	33
	34
Rationale. The definition of datatype equivalence implies that equivalent predefined	35
datatypes are equal. By requiring the same handle for named predefined datatypes,	36
it is possible to use the == or .EQ. comparison operator to determine the datatype involved. ( <i>End of rationale.</i> )	37
mvolved. (End of fationale.)	38
Advice to implementors. The datatypes returned in array_of_datatypes must appear	39
to the user as if each is an equivalent copy of the datatype used in the type constructor	40
call. Whether this is done by creating a new datatype or via another mechanism such	41
as a reference count mechanism is up to the implementation as long as the semantics	42
are preserved. (End of advice to implementors.)	43
	44
Rationale. The committed state and attributes of the returned datatype is delib-	45
erately left vague. The datatype used in the original construction may have been	46
modified since its use in the constructor call. Attributes can be added, removed, or	47 48
modified as well as having the datatype committed. The semantics given allow for	10

1 a reference count implementation without having to track these changes. (End of  $\mathbf{2}$ rationale.) 3 In the deprecated datatype constructor calls, the address arguments in Fortran are 4 of type INTEGER. In the preferred calls, the address arguments are of type 5INTEGER (KIND=MPI\_ADDRESS\_KIND). The call MPI\_TYPE\_GET\_CONTENTS returns all ad-6 dresses in an argument of type INTEGER(KIND=MPI\_ADDRESS\_KIND). This is true even if the 7 deprecated calls were used. Thus, the location of values returned can be thought of as being 8 returned by the C bindings. It can also be determined by examining the preferred calls for 9 datatype constructors for the deprecated calls that involve addresses. 10 11 Rationale. By having all address arguments returned in the 12array\_of\_addresses argument, the result from a C and Fortran decoding of a datatype 13 gives the result in the same argument. It is assumed that an integer of type 14INTEGER(KIND=MPI\_ADDRESS\_KIND) will be at least as large as the INTEGER argument 15used in datatype construction with the old MPI-1 calls so no loss of information will 16occur. (End of rationale.) 17 18 The following defines what values are placed in each entry of the returned arrays 19 depending on the datatype constructor used for datatype. It also specifies the size of the 20arrays needed which is the values returned by MPI\_TYPE\_GET\_ENVELOPE. In Fortran, 21the following calls were made: 22PARAMETER (LARGE = 1000) 23INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 24INTEGER(KIND=MPI\_ADDRESS\_KIND) A(LARGE) 25CONSTRUCT DATATYPE TYPE (NOT SHOWN) 1 26CALL MPI\_TYPE\_GET\_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR) 27IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN 28WRITE (\*, \*) "NI, NA, OR ND = ", NI, NA, ND, & 29 " RETURNED BY MPI\_TYPE\_GET\_ENVELOPE IS LARGER THAN LARGE = ", LARGE 30 CALL MPI\_ABORT(MPI\_COMM\_WORLD, 99, IERROR) 31ENDIF 32 CALL MPI\_TYPE\_GET\_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR) 33 34or in C the analogous calls of: 35 36 #define LARGE 1000 37 int ni, na, nd, combiner, i[LARGE]; 38MPI\_Aint a[LARGE]; 39 MPI\_Datatype type, d[LARGE]; 40/\* construct datatype type (not shown) \*/ 41 MPI\_Type\_get\_envelope(type, &ni, &na, &nd, &combiner); 42if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) { 43 fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd); 44fprintf(stderr, "MPI\_Type\_get\_envelope is larger than LARGE = %d\n", 45LARGE); 46MPI\_Abort(MPI\_COMM\_WORLD, 99); 47}; 48MPI\_Type\_get\_contents(type, ni, na, nd, i, a, d);

In the descriptions t	hat fo	ollow, the lower ca	se name of arguments is us	ed. <sup>1</sup>
If combiner is MPI_	If combiner is MPI_COMBINER_NAMED then it is erroneous to call			
MPI_TYPE_GET_CONT	ENTS			3
If combiner is MPI_COMBINER_DUP then				4
Constructor argument	С	Fortran location		5
oldtype	d[0]	D(1)		6
		D(1)	—	7
and $ni = 0$ , $na = 0$ , $nd =$			c ]	8
If combiner is MPI_C	'OMRI	NER_CONTIGUOU	5 then	9
Constructor argument	С	Fortran location	 L	10
count	i[0]	I(1)		11
oldtype	d[0]	D(1)		12
and $ni = 1$ , $na = 0$ , $nd =$	= 1.			13
If combiner is MPI_C		NER_VECTOR the	n	14
Constructor argument	С	Fortran location		15
count	$\frac{0}{i[0]}$	I(1)		16
		( )		17
blocklength stride	i[1]	I(2)		18
	i[2]	I(3)		19
oldtype	d[0]	D(1)	_	20
and $ni = 3$ , $na = 0$ , $nd =$				21
If combiner is MPI_C	OMBI	NER_HVECTOR th	len	22
Constructor argument	С	Fortran location	 L	23
count	i[0]	I(1)		24
blocklength	$\mathbf{i}[1]$	I(2)		25
stride	a[0]	A(1)		26
oldtype	d[0]	D(1)		27
and $ni = 2$ , $na = 1$ , $nd =$	= 1			28
If combiner is MPI_C		NER_INDEXED the	en	29 30
Constructor argument		С	Fortran location	31
count		i[0]	<u>I(1)</u>	32
array_of_blocklengths		i[1]  to  i[i[0]]	I(2) to $I(I(1)+1)$	33
array_of_displacements			I(I(1)+2) to $I(2*I(1)+1)$	34
oldtype	1110	d[0]	D(1)	35
	0.			36
and $ni = 2*count+1$ , na If combiner is MPI_C	· · ·		hor	37
II combiner is MPI_C				38
Constructor argument			tran location	39
count		i[0]	I(1)	40
$array_of_blocklengths$			to $I(I(1)+1)$	41
array_of_displacements	a[0]		) to $A(I(1))$	42
oldtype		d[0]	D(1)	43
and $ni = count+1$ , $na =$	count	$d_{\rm r},  {\rm nd} = 1.$		44
If combiner is MPI_C			OCK then	45
				46
				47

Constructor argument	С	Fortran location	
count	i[0]	I(1)	
blocklength	i[1]	I(2)	
array_of_displacements			
oldtype	d[0]	D(1)	
and $ni = count+2$ , $na = 0$	0, nd = 1.		
If combiner is MPI_C	OMBINER_HINDEX	$ED_BLOCK$ then	
Constructor argument	С	Fortran location	
count	i[0]	I(1)	
blocklength	i[1]	I(2)	
array_of_displacements oldtype	a[0] to a[i[0]-1] d[0]	$\begin{array}{c} A(1) \text{ to } A(I(1)) \\ D(1) \end{array}$	
		D(1)	
and $ni = 2$ , $na = count$ , n If combiner is MPI_C		$\Gamma$ then	
Constructor argument	С	Fortran location	
count	i[0]	I(1)	-
$array_of_blocklengths$	2 3	I(2) to $I(I(1)+1)$	
array_of_displacements	a[0] to $a[i[0]-1]$	A(1) to $A(I(1))$	
array_of_types	d[0]  to  d[i[0]-1]	D(1) to $D(I(1))$	-
and $ni = count+1$ , $na = count+1$	$\operatorname{count}, \operatorname{nd} = \operatorname{count}$		
If combiner is $MPI_C$	OMBINER_SUBAR	RAY then	
Constructor argument	С	Fortran	location
ndims	i[0]		(1)
array_of_sizes	i[1] to $i[i[0]]$		I(I(1)+1)
array_of_subsizes	i[i[0]+1] to $i[2*i]$		I(2*I(1)+1)
array_of_starts	i[2*i[0]+1] to $i[3*$		to $I(3*I(1)+1)$
order	i[3*i[0]+1]	,	(1)+2]
oldtype	d[0]	D	(1)
and $ni = 3*ndims+2$ , na	,	<	
If combiner is MPI_C			
Constructor argument	C :[0]	Forti	ran location
size	i[0]		I(1)
rank	i[1]		I(2)
ndims	i[2]	$\mathbf{n}$ $\mathbf{T}(\mathbf{A})$	I(3)
array_of_gsizes	i[3]  to  i[i[2] +		to $I(I(3)+3)$
array_of_distribs		[2]+2] I(I(3)+4)	
array_of_dargs		i[2]+2] I(2*I(3)+4)	
array_of_psizes	i[3*i[2]+3] to $i[4*i[2]+3]$		4) to $I(4*I(3)+3)$
order	i[4*i[2]+3]	1(4	4*I(3)+4)
oldtype	d[0]		D(1)
and $ni = 4*ndims+4$ , na	,		
If combiner is MPL C	OMBINER_F90_RE	AL then	
II comonici is witi_c			

Constructor argument C Fortran location	1
$\frac{1}{p} \frac{i[0]}{I(1)}$	2
$\mathbf{r}$ $\mathbf{i}[1]$ $\mathbf{I}(2)$	3
	4
and $ni = 2$ , $na = 0$ , $nd = 0$ .	5
If combiner is MPI_COMBINER_F90_COMPLEX then	6
Constructor argument C Fortran location	7
p $i[0]$ $I(1)$	8
r $i[1]$ $I(2)$	9
and $ni = 2$ , $na = 0$ , $nd = 0$ .	10
If combiner is MPI_COMBINER_F90_INTEGER then	11
Constructor argument C Fortran location	12
$\frac{1}{r} \frac{i[0]}{I(1)}$	13
	14
and $ni = 1$ , $na = 0$ , $nd = 0$ .	15
If combiner is MPI_COMBINER_RESIZED then	16
Constructor argument C Fortran location	17
1b $a[0]$ $A(1)$	18
extent $a[1]$ $A(2)$	19
oldtype $d[0] D(1)$	20
and $ni = 0$ , $na = 2$ , $nd = 1$ .	21
	22
4.1.14 Examples	23
	24
The following examples illustrate the use of derived datatypes.	25
Francisco 4 19 Candender a section of a 2D among	26
<b>Example 4.13</b> Send and receive a section of a 3D array.	27
REAL a(100,100,100), e(9,9,9)	28
INTEGER oneslice, twoslice, threeslice, myrank, ierr	29
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal	30
INTEGER status(MPI_STATUS_SIZE)	31
	32
C extract the section a(1:17:2, 3:11, 2:10)	33
C and store it in $e(:,:,:)$ .	34
	35
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)	36
OREL IN I_OUNI_RAWR(IN I_OUNI_WORLD, myrawr, Ieir)	37
CALL MPI_TYPE_GET_EXTENT( MPI_REAL, lb, sizeofreal, ierr)	38
	39
C create datatype for a 1D section	40
CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)	41
	42
C create datatype for a 2D section	43
CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twosl	ice, ierr)
and miling on and method of the second of th	40
C create datatype for the entire section	46
CALL MPI_TYPE_CREATE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice,	47
and in t_int_ownit_intoion( ), i, ioutioursizeditedit, buosilee,	48

1		threeslice, ierr)
3 4 5 6		CALL MPI_TYPE_COMMIT( threeslice, ierr) CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
7 8	Exan	<b>aple 4.14</b> Copy the (strictly) lower triangular part of a matrix.
9 10 11 12		REAL a(100,100), b(100,100) INTEGER disp(100), blocklen(100), ltype, myrank, ierr INTEGER status(MPI_STATUS_SIZE)
13 14 15	C C	copy lower triangular part of array a onto lower triangular part of array b
16 17		CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
18 19 20 21 22 23	С	<pre>compute start and size of each column DO i=1, 100   disp(i) = 100*(i-1) + i   blocklen(i) = 100-i END DO</pre>
24 25 26	С	create datatype for lower triangular part CALL MPI_TYPE_INDEXED( 100, blocklen, disp, MPI_REAL, ltype, ierr)
27 28 29 30		CALL MPI_TYPE_COMMIT(ltype, ierr) CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1, ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
31 32	Exan	<b>aple 4.15</b> Transpose a matrix.
33 34 35 36 37		REAL a(100,100), b(100,100) INTEGER row, xpose, myrank, ierr INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal INTEGER status(MPI_STATUS_SIZE)
38 39	С	transpose matrix a onto b
40 41		CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
42 43		CALL MPI_TYPE_GET_EXTENT( MPI_REAL, lb, sizeofreal, ierr)
44 45 46	С	create datatype for one row CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
47 48	С	create datatype for matrix in row-major order

```
CALL MPI_TYPE_CREATE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)
                                                                                    1
                                                                                    \mathbf{2}
                                                                                    3
      CALL MPI_TYPE_COMMIT( xpose, ierr)
                                                                                    4
                                                                                    5
С
      send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
                                                                                    6
                                                                                    7
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
                                                                                    8
                                                                                    9
Example 4.16 Another approach to the transpose problem:
                                                                                    10
                                                                                   11
      REAL a(100,100), b(100,100)
                                                                                   12
      INTEGER row, row1
                                                                                   13
      INTEGER (KIND=MPI_ADDRESS_KIND) disp(2), lb, sizeofreal
                                                                                   14
      INTEGER myrank, ierr
                                                                                   15
      INTEGER status(MPI_STATUS_SIZE)
                                                                                   16
                                                                                   17
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
                                                                                   18
                                                                                   19
С
      transpose matrix a onto b
                                                                                   20
                                                                                   21
      CALL MPI_TYPE_GET_EXTENT( MPI_REAL, lb, sizeofreal, ierr)
                                                                                   22
                                                                                   23
С
      create datatype for one row
                                                                                   24
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
                                                                                   25
                                                                                   26
С
      create datatype for one row, with the extent of one real number
                                                                                   27
      1b = 0
                                                                                   28
      CALL MPI_TYPE_CREATE_RESIZED( row, lb, sizeofreal, row1, ierr)
                                                                                   29
                                                                                   30
      CALL MPI_TYPE_COMMIT( row1, ierr)
                                                                                   31
                                                                                   32
С
      send 100 rows and receive in column major order
                                                                                   33
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                                                                                   34
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
                                                                                   35
                                                                                   36
                                                                                   37
Example 4.17 We manipulate an array of structures.
                                                                                   38
                                                                                   39
struct Partstruct
                                                                                    40
   ſ
                                                                                   41
             class; /* particle class */
      int
                                                                                   42
      double d[6]; /* particle coordinates */
      char b[7]; /* some additional information */
                                                                                   43
                                                                                   44
   };
                                                                                   45
                                                                                    46
struct Partstruct particle[1000];
                                                                                    47
                                                                                    48
int
             i, dest, tag;
```

```
1
     MPI_Comm
                   comm;
\mathbf{2}
3
4
     /* build datatype describing structure */
\mathbf{5}
6
     MPI_Datatype Particlestruct, Particletype;
7
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
8
                   blocklen[3] = \{1, 6, 7\};
     int
9
     MPI_Aint
                   disp[3];
10
     MPI_Aint
                   base, lb, sizeofentry;
11
12
13
     /* compute displacements of structure components */
14
15
     MPI_Get_address( particle, disp );
16
     MPI_Get_address( particle[0].d, disp+1 );
17
     MPI_Get_address( particle[0].b, disp+2 );
18
     base = disp[0];
19
     for (i=0; i < 3; i++) disp[i] -= base;</pre>
20
21
     MPI_Type_create_struct( 3, blocklen, disp, type, &Particlestruct);
22
23
        /* If compiler does padding in mysterious ways,
^{24}
        the following may be safer */
25
26
     /* compute extent of the structure */
27
28
     MPI_Get_address( particle+1, &sizeofentry );
29
     sizeofentry -= base;
30
^{31}
     /* build datatype describing structure */
32
33
     MPI_Type_create_resized( Particlestruct, 0, sizeofentry, &Particletype );
34
35
36
                    /* 4.1:
37
              send the entire array */
38
39
     MPI_Type_commit( &Particletype);
40
     MPI_Send( particle, 1000, Particletype, dest, tag, comm);
41
42
43
                    /* 4.2:
44
              send only the entries of class zero particles,
45
             preceded by the number of such entries */
46
47
     MPI_Datatype Zparticles;
                                  /* datatype describing all particles
48
                                     with class zero (needs to be recomputed
```

```
1
                                 if classes change) */
                                                                                       \mathbf{2}
MPI_Datatype Ztype;
                                                                                       3
                                                                                       4
int
              zdisp[1000];
int
              zblock[1000], j, k;
                                                                                       5
                                                                                       6
int
              zzblock[2] = {1,1};
                                                                                       7
MPI_Aint
              zzdisp[2];
                                                                                        8
MPI_Datatype zztype[2];
                                                                                       9
/* compute displacements of class zero particles */
                                                                                       10
                                                                                       11
j = 0;
for(i=0; i < 1000; i++)</pre>
                                                                                       12
   if (particle[i].class == 0)
                                                                                       13
                                                                                       14
      {
                                                                                       15
        zdisp[j] = i;
                                                                                       16
        zblock[j] = 1;
                                                                                       17
        j++;
      }
                                                                                       18
                                                                                       19
                                                                                       20
/* create datatype for class zero particles */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
                                                                                       21
                                                                                       22
/* prepend particle count */
                                                                                       23
                                                                                       ^{24}
MPI_Get_address(&j, zzdisp);
                                                                                       25
MPI_Get_address(particle, zzdisp+1);
                                                                                       26
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
                                                                                       27
MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
                                                                                       28
                                                                                       29
                                                                                       30
MPI_Type_commit( &Ztype);
                                                                                       31
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
                                                                                       32
                                                                                       33
                                                                                       34
       /* A probably more efficient way of defining Zparticles */
                                                                                       35
/* consecutive particles with index zero are handled as one block */
                                                                                       36
                                                                                       37
j=0;
for (i=0; i < 1000; i++)</pre>
                                                                                       38
                                                                                       39
   if (particle[i].class == 0)
                                                                                       40
      {
                                                                                       41
          for (k=i+1; (k < 1000)&&(particle[k].class == 0) ; k++);</pre>
                                                                                       42
          zdisp[j] = i;
          zblock[j] = k-i;
                                                                                       43
                                                                                       44
          j++;
                                                                                       45
          i = k;
                                                                                       46
      }
                                                                                       47
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
                                                                                       48
```

1

```
\mathbf{2}
                      /* 4.3:
3
                send the first two coordinates of all entries */
4
\mathbf{5}
     MPI_Datatype Allpairs;
                              /* datatype for all pairs of coordinates */
6
7
     MPI_Type_get_extent( Particletype, &lb, &sizeofentry);
8
9
          /* sizeofentry can also be computed by subtracting the address
10
              of particle[0] from the address of particle[1] */
11
12
     MPI_Type_create_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
13
     MPI_Type_commit( &Allpairs);
14
     MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);
15
16
           /* an alternative solution to 4.3 */
17
18
     MPI_Datatype Twodouble;
19
     MPI_Type_contiguous( 2, MPI_DOUBLE, &Twodouble);
20
21
22
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
23
                                 the extent of one particle entry */
^{24}
25
     MPI_Type_create_resized( Twodouble, 0, sizeofentry, &Onepair );
26
     MPI_Type_commit( &Onepair);
27
     MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
28
29
30
     Example 4.18 The same manipulations as in the previous example, but use absolute
31
     addresses in datatypes.
32
33
     struct Partstruct
34
        {
35
           int class;
36
           double d[6];
37
           char b[7];
38
        };
39
40
     struct Partstruct particle[1000];
41
42
                 /* build datatype describing first array entry */
43
44
     MPI_Datatype Particletype;
45
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
46
                   block[3] = \{1, 6, 7\};
     int
47
     MPI_Aint
                   disp[3];
48
```

```
\mathbf{2}
MPI_Get_address( particle, disp);
                                                                                      3
MPI_Get_address( particle[0].d, disp+1);
MPI_Get_address( particle[0].b, disp+2);
                                                                                      4
MPI_Type_create_struct( 3, block, disp, type, &Particletype);
                                                                                      5
                                                                                      6
/* Particletype describes first array entry -- using absolute
                                                                                      7
                                                                                      8
   addresses */
                                                                                      9
                                                                                      10
                   /* 5.1:
                                                                                      11
             send the entire array */
                                                                                      12
MPI_Type_commit( &Particletype);
                                                                                      13
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
                                                                                      14
                                                                                      15
                                                                                      16
                                                                                      17
                  /* 5.2:
                                                                                      18
          send the entries of class zero,
                                                                                      19
          preceded by the number of such entries */
                                                                                      20
                                                                                      21
MPI_Datatype Zparticles, Ztype;
                                                                                      22
                                                                                      23
              zdisp[1000];
int
                                                                                      24
              zblock[1000], i, j, k;
int
                                                                                      25
int
              zzblock[2] = {1,1};
                                                                                      26
MPI_Datatype zztype[2];
MPI_Aint
              zzdisp[2];
                                                                                      27
                                                                                      28
                                                                                      29
j=0;
                                                                                      30
for (i=0; i < 1000; i++)
                                                                                      31
   if (particle[i].class == 0)
      {
                                                                                      32
                                                                                      33
         for (k=i+1; (k < 1000)&&(particle[k].class == 0) ; k++);</pre>
                                                                                      34
         zdisp[j] = i;
         zblock[j] = k-i;
                                                                                      35
                                                                                      36
          j++;
                                                                                      37
          i = k;
                                                                                      38
      }
                                                                                      39
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
                                                                                      40
                                                                                      41
   their absolute addresses*/
                                                                                      42
/* prepend particle count */
                                                                                      43
                                                                                      44
MPI_Get_address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
                                                                                      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
5
6
     Example 4.19 Handling of unions.
7
8
     union {
9
        int
                 ival;
10
        float
                 fval;
11
            } u[1000];
12
13
     int
              utype;
14
15
     /* All entries of u have identical type; variable
16
        utype keeps track of their current type */
17
18
     MPI_Datatype
                      mpi_utype[2];
19
     MPI_Aint
                      i, extent;
20
21
     /* compute an MPI datatype for each possible union type;
22
        assume values are left-aligned in union storage. */
23
24
     MPI_Get_address( u, &i);
25
     MPI_Get_address( u+1, &extent);
26
     extent -= i;
27
28
     MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
29
30
     MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
^{31}
32
     for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
33
34
     /* actual communication */
35
36
     MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
37
38
39
     Example 4.20 This example shows how a datatype can be decoded. The routine
40
     printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
41
     datatypes that are not predefined.
42
     /*
43
       Example of decoding a datatype.
44
45
46
       Returns 0 if the datatype is predefined, 1 otherwise
47
      */
     #include <stdio.h>
48
```

{

```
1
#include <stdlib.h>
                                                                                    2
#include "mpi.h"
                                                                                    3
int printdatatype( MPI_Datatype datatype )
                                                                                    4
    int *array_of_ints;
                                                                                    5
                                                                                    6
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
                                                                                    7
    int num_ints, num_adds, num_dtypes, combiner;
                                                                                    9
    int i;
                                                                                    10
                                                                                    11
    MPI_Type_get_envelope( datatype,
                            &num_ints, &num_adds, &num_dtypes, &combiner );
                                                                                    12
    switch (combiner) {
                                                                                    13
                                                                                    14
    case MPI_COMBINER_NAMED:
                                                                                    15
        printf( "Datatype is named:" );
                                                                                    16
        /* To print the specific type, we can match against the
                                                                                    17
           predefined forms. We can NOT use a switch statement here
                                                                                    18
           We could also use MPI_TYPE_GET_NAME if we prefered to use
                                                                                    19
           names that the user may have changed.
         */
                                                                                    20
                                                                                    21
        if
                 (datatype == MPI_INT)
                                           printf( "MPI_INT\n" );
        else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
                                                                                    22
                                                                                    23
        ... else test for other types ...
                                                                                    24
        return 0;
                                                                                    25
        break;
                                                                                    26
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
                                                                                    27
                                                                                    28
        printf( "Datatype is struct containing" );
                                                                                    29
        array_of_ints
                        = (int *)malloc( num_ints * sizeof(int) );
        array_of_adds
                                                                                    30
                                                                                    31
                    (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
                                                                                    32
        array_of_dtypes = (MPI_Datatype *)
                                                                                    33
            malloc( num_dtypes * sizeof(MPI_Datatype) );
                                                                                    34
        MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
                          array_of_ints, array_of_adds, array_of_dtypes );
                                                                                    35
        printf( " %d datatypes:\n", array_of_ints[0] );
                                                                                    36
                                                                                    37
        for (i=0; i<array_of_ints[0]; i++) {</pre>
                                                                                    38
            printf( "blocklength %d, displacement %ld, type:\n",
                                                                                    39
                     array_of_ints[i+1], array_of_adds[i] );
            if (printdatatype( array_of_dtypes[i] )) {
                                                                                    40
                                                                                    41
                 /* Note that we free the type ONLY if it
                                                                                    42
                    is not predefined */
                 MPI_Type_free( &array_of_dtypes[i] );
                                                                                    43
                                                                                    44
            }
                                                                                    45
        }
                                                                                    46
        free( array_of_ints );
                                                                                    47
        free( array_of_adds );
                                                                                    48
        free( array_of_dtypes );
```

```
break;
break;
... other combiner values ...
default:
printf( "Unrecognized combiner type\n" );
}
return 1;
}
```

```
4.2 Pack and Unpack
```

11Some existing communication libraries provide pack/unpack functions for sending noncon-12tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending 13 it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are 14described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. 15The user specifies the layout of the data to be sent or received, and the communication 16library directly accesses a noncontiguous buffer. The pack/unpack routines are provided 17for compatibility with previous libraries. Also, they provide some functionality that is not 18 otherwise available in MPI. For instance, a message can be received in several parts, where 19the receive operation done on a later part may depend on the content of a former part. 20Another use is that outgoing messages may be explicitly buffered in user supplied space, 21thus overriding the system buffering policy. Finally, the availability of pack and unpack 22operations facilitates the development of additional communication libraries layered on top 23of MPI. 24

```
25
26
```

9

10

MPI\_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

			,
27 28	IN	inbuf	input buffer start (choice)
29	IN	incount	number of input data items (non-negative integer)
30	IN	datatype	datatype of each input data item (handle)
31	OUT	outbuf	output buffer start (choice)
32 33	IN	outsize	output buffer size, in bytes (non-negative integer)
34	INOUT	position	current position in buffer, in bytes (integer)
35	IN	comm	communicator for packed message (handle)
36			
37			the second MDT Determine determine
38	int MPI_P		it incount, MPI_Datatype datatype,
39		void *outbuf, int ou	tsize, int *position, MPI_Comm comm)
40	MPI_Pack(	inbuf, incount, datatype,	outbuf, outsize, position, comm, ierror)
41		BIND(C)	
42	TYPE(	*), DIMENSION(), INTENT	S(IN) :: inbuf
43	TYPE(	*), DIMENSION() :: ou	ıtbuf
44	INTEG	ER, INTENT(IN) :: incour	nt, outsize
45	TYPE(	MPI_Datatype), INTENT(IN)	:: datatype
46	INTEG	ER, INTENT(INOUT) :: pos	sition
47	TYPE(	MPI_Comm), INTENT(IN) ::	comm
48	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror

MPI\_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR) <type> INBUF(\*), OUTBUF(\*) INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR

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.

```
16
MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
                                                                                          17
  IN
           inbuf
                                       input buffer start (choice)
                                                                                          18
  IN
           insize
                                       size of input buffer, in bytes (non-negative integer)
                                                                                          19
                                                                                          20
  INOUT
           position
                                       current position in bytes (integer)
                                                                                          21
           outbuf
  OUT
                                       output buffer start (choice)
                                                                                          22
  IN
           outcount
                                       number of items to be unpacked (integer)
                                                                                          23
                                                                                          ^{24}
  IN
           datatype
                                       datatype of each output data item (handle)
                                                                                          25
  IN
           comm
                                       communicator for packed message (handle)
                                                                                          26
                                                                                          27
int MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf,
                                                                                          28
               int outcount, MPI_Datatype datatype, MPI_Comm comm)
                                                                                          29
                                                                                          30
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
                                                                                          31
               ierror) BIND(C)
                                                                                          32
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                          33
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                          34
    INTEGER, INTENT(IN) :: insize, outcount
                                                                                          35
    INTEGER, INTENT(INOUT) :: position
                                                                                          36
    TYPE(MPI_Datatype), INTENT(IN) ::
                                            datatype
                                                                                          37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          39
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
                                                                                          40
               IERROR)
                                                                                          41
    <type> INBUF(*), OUTBUF(*)
                                                                                          42
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
                                                                                          43
                                                                                          44
    Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from
```

45the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI\_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

1415

46

47

<sup>1</sup> buffer occupied by the packed message. **position** is incremented by the size of the packed <sup>2</sup> message, so that the output value of **position** is the first location in the input buffer after <sup>3</sup> the locations occupied by the message that was unpacked. **comm** is the communicator used <sup>4</sup> to receive the packed message.

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.)

<sup>16</sup> To understand the behavior of pack and unpack, it is convenient to think of the data <sup>17</sup> part of a message as being the sequence obtained by concatenating the successive values sent <sup>19</sup> in that message. The pack operation stores this sequence in the buffer space, as if sending <sup>20</sup> the message to that buffer. The unpack operation retrieves this sequence from buffer space, <sup>21</sup> as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or <sup>22</sup> 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

<sup>37</sup> MPI\_UNPACK, where the first call provides position = 0, and each successive call inputs the <sup>38</sup> value of position that was output by the previous call, and the same values for inbuf, insize <sup>40</sup> 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,

## Unofficial Draft for Comment Only

5

6

7

8

9

10

11

12

13

14

15

34

35

{

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.

MPI\_PACK\_SIZE(incount, datatype, comm, size)

MPI_P	ACK_SIZE(incount, datatype	e, comm, size)	8
IN	incount	count argument to packing call (non-negative integer)	9
IN	datatype	datatype argument to packing call (handle)	10
IN	comm	communicator argument to packing call (handle)	11 12
OUT	size	upper bound on size of packed message, in bytes (non-	13
		negative integer)	14
			15
int M		z, MPI_Datatype datatype, MPI_Comm comm,	16 17
	int *size)		18
MPI_Pa	ack_size(incount, dataty	<pre>rpe, comm, size, ierror) BIND(C)</pre>	19
	•	ncount	20
	YPE(MPI_Datatype), INTEN		21
	YPE(MPI_Comm), INTENT(IN NTEGER, INTENT(OUT) ::		22
	NTEGER, OPTIONAL, INTENI		23 24
			25
	ACK_SIZE(INCOUNT, DATATY NTEGER INCOUNT, DATATYPE		26
			27
		bunt, datatype, comm, size) returns in size an upper bound effected by a call to MPI_PACK(inbuf, incount, datatype,	28
	-	If the packed size of the datatype cannot be expressed	29 30
		PACK_SIZE sets the value of size to MPI_UNDEFINED.	31
Ū			32
		an upper bound, rather than an exact bound, since the	33
	-	d to pack the message may depend on the context (e.g., king unit may take more space). ( <i>End of rationale.</i> )	34
1	iist message packed in a pac	king unit may take more space). (End of futtonale.)	35
			36 37
Exam	ple 4.21 An example using	g MPI_PACK.	38
int	position, i, j, a	[2];	39
char	buff[1000];		40
			41
	omm_rank(MPI_COMM_WORLD,	&myrank);	42
ıf (my	yrank == 0)		43

/\* SENDER CODE \*/ position = 0; MPI\_Pack(&i, 1, MPI\_INT, buff, 1000, &position, MPI\_COMM\_WORLD);

**Unofficial Draft for Comment Only** 

1 $\mathbf{2}$ 

3

4

56  $\overline{7}$ 

44

4546

47

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

```
{
                                                                                       1
                                                                                       \mathbf{2}
   /* RECEIVER CODE */
                                                                                       3
  MPI_Status status;
                                                                                       4
                                                                                       5
  /* Receive */
                                                                                       6
                                                                                       7
                                                                                       8
  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
                                                                                       9
  /* Unpack i */
                                                                                       10
                                                                                       11
  position = 0;
                                                                                       12
  MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
                                                                                       13
                                                                                       14
                                                                                       15
  /* Unpack a[0]...a[i-1] */
  MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
                                                                                       16
                                                                                       17
}
                                                                                       18
                                                                                       19
Example 4.23 Each process sends a count, followed by count characters to the root; the
                                                                                       20
root concatenates all characters into one string.
                                                                                       21
int count, gsize, counts[64], totalcount, k1, k2, k,
                                                                                       22
                                                                                       23
     displs[64], position, concat_pos;
                                                                                       ^{24}
char chr[100], *lbuf, *rbuf, *cbuf;
                                                                                       25
                                                                                       26
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
                                                                                       27
                                                                                       28
                                                                                       29
      /* allocate local pack buffer */
                                                                                       30
MPI_Pack_size(1, MPI_INT, comm, &k1);
                                                                                       31
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
                                                                                       32
k = k1+k2;
                                                                                       33
lbuf = (char *)malloc(k);
                                                                                       34
      /* pack count, followed by count characters */
                                                                                       35
                                                                                       36
position = 0;
                                                                                       37
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
```

MPI\_Pack(chr, count, MPI\_CHAR, lbuf, k, &position, comm);

```
1
     } else {
                 /* root code */
\mathbf{2}
            /* gather sizes of all packed messages */
3
        MPI_Gather( &position, 1, MPI_INT, counts, 1,
4
                   MPI_INT, root, comm);
5
6
            /* gather all packed messages */
7
        displs[0] = 0;
8
        for (i=1; i < gsize; i++)</pre>
9
          displs[i] = displs[i-1] + counts[i-1];
10
        totalcount = displs[gsize-1] + counts[gsize-1];
11
        rbuf = (char *)malloc(totalcount);
12
        cbuf = (char *)malloc(totalcount);
13
        MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
14
                  counts, displs, MPI_PACKED, root, comm);
15
16
             /* unpack all messages and concatenate strings */
17
        concat_pos = 0;
18
        for (i=0; i < gsize; i++) {</pre>
19
           position = 0;
20
           MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
21
                  &position, &count, 1, MPI_INT, comm);
22
           MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
23
                  &position, cbuf+concat_pos, count, MPI_CHAR, comm);
24
            concat_pos += count;
25
        }
26
        cbuf[concat_pos] = ' \ ';
27
     }
28
```

# 4.3 Canonical MPI\_PACK and MPI\_UNPACK

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

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

The buffer will contain exactly the packed data, without headers. MPI\_BYTE should be used to send and receive data that is packed using MPI\_PACK\_EXTERNAL.

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

29

30 31

32

33

34

35 36

37

38

39

40 41

42

43

44

MPI_PACK	<pre>C_EXTERNAL(datarep, inbuf,</pre>	, incount, datatype, outbuf, outsize, position)	1
IN	datarep	data representation (string)	2 3
IN	inbuf	input buffer start (choice)	4
IN	incount	number of input data items (integer)	5
IN	datatype	datatype of each input data item (handle)	6
OUT	outbuf	output buffer start (choice)	7 8
IN	outsize	output buffer size, in bytes (integer)	9
INOUT	position	current position in buffer, in bytes (integer)	10 11
int MPI_F		<pre>datarep[], const void *inbuf, int incount, ype, void *outbuf, MPI_Aint outsize,</pre>	12 13 14 15
MPI_Pack_	-	, incount, datatype, outbuf, outsize,	16
<b>011 A D A</b>	position, ierror) B		17
	ACTER(LEN=*), INTENT(IN) (*), DIMENSION(), INTE	-	18 19
	(*), DIMENSION() ::		20
	ER, INTENT(IN) :: inco		21
	MPI_Datatype), INTENT(I		22
		D), INTENT(IN) :: outsize D), INTENT(INOUT) :: position	23 24
	ER, OPTIONAL, INTENT(OU	-	24
			26
MPI_PACK_	POSITION, IERROR)	, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	27
INTEG	ER INCOUNT, DATATYPE, I	ERROR	28 29
INTEG	ER(KIND=MPI_ADDRESS_KIN	D) OUTSIZE, POSITION	29 30
	CTER*(*) DATAREP		31
<type< td=""><td><pre>&gt; INBUF(*), OUTBUF(*)</pre></td><td></td><td>32</td></type<>	<pre>&gt; INBUF(*), OUTBUF(*)</pre>		32
			33
MPI_UNPA	ACK_EXTERNAL(datarep, in	buf, insize, position, outbuf, outsize, position)	34 35
IN	datarep	data representation (string)	36
IN	inbuf	input buffer start (choice)	37
IN	insize	input buffer size, in bytes (integer)	38
INOUT	position	current position in buffer, in bytes (integer)	39 40
OUT	outbuf	- · · · · · · · · · · · · · · · · · · ·	41
		output buffer start (choice)	42
IN	outcount	number of output data items (integer)	43
IN	datatype	datatype of output data item (handle)	44
int MDT T	Innack external (const ch	ar datarep[], const void *inbuf,	45 46
IIIC MFI_U	•	PI_Aint *position, void *outbuf,	47
	int outcount, MPI_D	-	48

```
1
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
\mathbf{2}
                   datatype, ierror) BIND(C)
3
         CHARACTER(LEN=*), INTENT(IN) :: datarep
4
         TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
5
         TYPE(*), DIMENSION(..) :: outbuf
6
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
7
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
8
         INTEGER, INTENT(IN) :: outcount
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                               ierror
11
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
12
                   DATATYPE, IERROR)
13
         INTEGER OUTCOUNT, DATATYPE, IERROR
14
         INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
15
         CHARACTER*(*) DATAREP
16
         <type> INBUF(*), OUTBUF(*)
17
18
19
     MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
20
21
       IN
                datarep
                                           data representation (string)
22
       IN
                incount
                                           number of input data items (integer)
23
       IN
                                           datatype of each input data item (handle)
                datatype
24
25
       OUT
                size
                                           output buffer size, in bytes (integer)
26
27
     int MPI_Pack_external_size(const char datarep[], int incount,
28
                   MPI_Datatype datatype, MPI_Aint *size)
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
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
37
         INTEGER INCOUNT, DATATYPE, IERROR
38
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
39
         CHARACTER*(*) DATAREP
40
41
42
43
44
45
46
47
48
```

# Chapter 5

# **Collective Communication**

## 5.1 Introduction and Overview

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

- MPI\_BARRIER, MPI\_IBARRIER: Barrier synchronization across all members of a group (Section 5.3 and Section 5.12.1).
- MPI\_BCAST, MPI\_IBCAST: Broadcast from one member to all members of a group (Section 5.4 and Section 5.12.2). This is shown as "broadcast" in Figure 5.1.
- MPI\_GATHER, MPI\_IGATHER, MPI\_GATHERV, MPI\_IGATHERV: Gather data from all members of a group to one member (Section 5.5 and Section 5.12.3). This is shown as "gather" in Figure 5.1.
- MPI\_SCATTER, MPI\_ISCATTER, MPI\_SCATTERV, MPI\_ISCATTERV: Scatter data from one member to all members of a group (Section 5.6 and Section 5.12.4). This is shown as "scatter" in Figure 5.1.
- MPI\_ALLGATHER, MPI\_IALLGATHER, MPI\_ALLGATHERV, MPI\_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7 and Section 5.12.5). This is shown as "allgather" in Figure 5.1.
- MPI\_ALLTOALL, MPI\_IALLTOALL, MPI\_ALLTOALLV, MPI\_IALLTOALLV, MPI\_ALLTOALLW, MPI\_IALLTOALLW, MPI\_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 5.8 and Section 5.12.6). This is shown as "complete exchange" in Figure 5.1.
- MPI\_ALLREDUCE, MPI\_IALLREDUCE, MPI\_REDUCE, MPI\_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 5.9.6 and Section 5.12.8) and a variation where the result is returned to only one member (Section 5.9 and Section 5.12.7).
- MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_IREDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER, MPI\_IREDUCE\_SCATTER: A combined reduction and scatter operation (Section 5.10, Section 5.12.9, and Section 5.12.10).

1 2

3

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 24to 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.

(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.

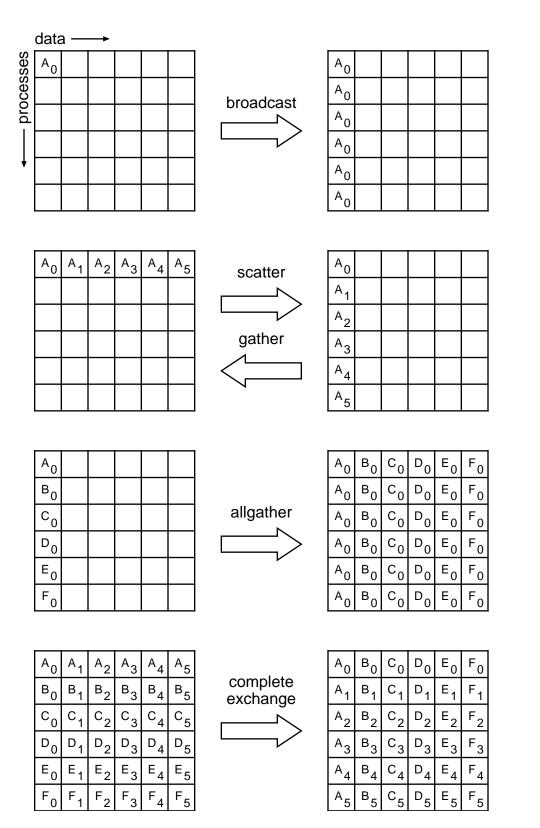


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data  $A_0$ , but after the broadcast all processes contain it.

 $\mathbf{2}$ 

 $^{31}$ 

 $^{41}$ 

 $44 \\ 45$ 

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.

## Unofficial Draft for Comment Only

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, [57]):

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

<ul> <li>MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV,</li> </ul>
MPI_IALLGATHERV
• MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV,

- MPI\_ALLTOALLW, MPI\_IALLTOALLWMPI\_ALLREDUCE, MPI\_IALLREDUCE, MPI\_REDUCE\_SCATTER\_BLOCK,
- MPI\_IREDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER, MPI\_IREDUCE\_SCATTER
- MPI\_BARRIER, MPI\_IBARRIER

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

- MPI\_GATHER, MPI\_IGATHER, MPI\_GATHERV, MPI\_IGATHERV
- MPI\_REDUCE, MPI\_IREDUCE

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

- MPI\_BCAST, MPI\_IBCAST
- MPI\_SCATTER, MPI\_ISCATTER, MPI\_SCATTERV, MPI\_ISCATTERV

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

• MPI\_SCAN, MPI\_ISCAN, MPI\_EXSCAN, MPI\_IEXSCAN

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

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

The following collective operations also apply to intercommunicators:

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

#### Unofficial Draft for Comment Only

1 2

3 4

5

6

7

8 9

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

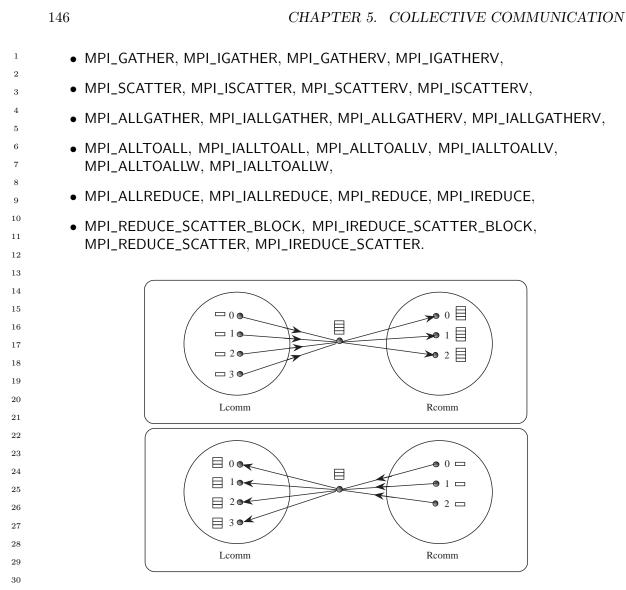


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

33 34 35

36

 $^{31}$ 

32

## 5.2.3 Specifics for Intercommunicator Collective Operations

 $_{_{37}}$  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

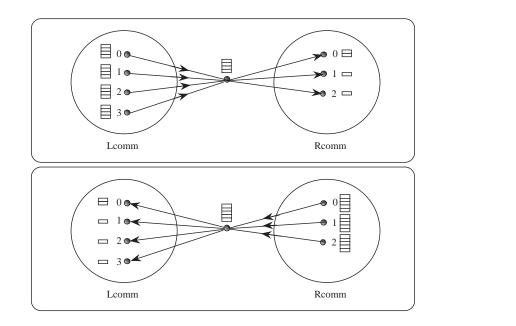


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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.

#### Unofficial Draft for Comment Only

 $44 \\ 45$ 

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.

## 5.4 Broadcast

```
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)

```
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)
Stype> BUFFER(\*)
INTECEP COUNT DATATYPE POOT COMM IERROP

```
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

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

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

The "in place" option is not meaningful here.

<sup>42</sup> If comm is an intercommunicator, then the call involves all processes in the intercom-<sup>43</sup> municator, but with one group (group A) defining the root process. All processes in the <sup>44</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>46</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>47</sup> pass the value MPI\_PROC\_NULL in root. Data is broadcast from the root to all processes <sup>48</sup>

1

 $\mathbf{2}$ 

3

4

5 6

20 21

22

23

24

25

26

1 in group B. The buffer arguments of the processes in group B must be consistent with the 2 buffer argument of the root. 3 4 5.4.1 Example using MPI\_BCAST 5The examples in this section use intracommunicators. 6 7 Example 5.1 8 Broadcast 100 ints from process 0 to every process in the group. 9 10 MPI\_Comm comm; 11 int array[100]; 12int root=0; 13 . . . 14MPI\_Bcast(array, 100, MPI\_INT, root, comm); 1516As in many of our example code fragments, we assume that some of the variables (such as 17comm in the above) have been assigned appropriate values. 18 19 5.5 Gather 202122 23MPI\_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)  $^{24}$ IN sendbuf starting address of send buffer (choice) 2526IN sendcount number of elements in send buffer (non-negative integer) 2728IN sendtype data type of send buffer elements (handle) 29OUT recvbuf address of receive buffer (choice, significant only at 30 root) 31IN recvcount number of elements for any single receive (non-negative 32 33 integer, significant only at root) 34 IN recvtype data type of recv buffer elements (significant only at 35root) (handle) 36 IN rank of receiving process (integer) root 37 IN communicator (handle) 38 comm

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

Unofficial Draft for Comment Only

39 40

41

42

43

44

45

46

47

1 2	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
5	ROOT, COMM, IERROR)
6	<type> SENDBUF(*), RECVBUF(*)</type>
7 °	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
8 9	
10	If comm is an intracommunicator, each process (root process included) sends the con-
11	tents of its send buffer to the root process. The root process receives the messages and stores
12	them in rank order. The outcome is as if each of the n processes in the group (including the next process) had executed a call to
13	the root process) had executed a call to
14	$\texttt{MPI}\_\texttt{Send}(\texttt{sendbuf},\texttt{sendcount},\texttt{sendtype},\texttt{root},),$
15	and the root had executed n calls to
16	and the root had executed if cans to
17	$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{i} \cdot \texttt{recvcount} \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcount}, \texttt{recvtype}, \texttt{i},),$
18	
19	where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent(). An alternative description is that the n messages sent by the processes in the group
20	are concatenated in rank order, and the resulting message is received by the root as if by a
21	call to MPI_RECV(recvbuf, recvcount $\cdot$ n, recvtype,).
22	The receive buffer is ignored for all non-root processes.
23 24	General, derived datatypes are allowed for both sendtype and recvtype. The type signa-
24 25	ture of sendcount, sendtype on each process must be equal to the type signature of recvcount,
26	recvtype at the root. This implies that the amount of data sent must be equal to the amount
20	of data received, pairwise between each process and the root. Distinct type maps between
28	sender and receiver are still allowed.
29	All arguments to the function are significant on process <b>root</b> , while on other processes,
30	only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
31	root and comm must have identical values on all processes.
32	The specification of counts and types should not cause any location on the root to be
33	written more than once. Such a call is erroneous.
34	Note that the <b>recvcount</b> argument at the root indicates the number of items it receives
35	from <i>each</i> process, not the total number of items it receives.
36	The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as
37	the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and
38	the contribution of the root to the gathered vector is assumed to be already in the correct
39	place in the receive buffer.
40	If comm is an intercommunicator, then the call involves all processes in the intercom-
41	municator, but with one group (group A) defining the root process. All processes in the
42	other group (group B) pass the same value in argument root, which is the rank of the root
43	in group A. The root passes the value MPI_ROOT in root. All other processes in group A
44	pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to
45	the root. The send buffer arguments of the processes in group B must be consistent with
46	the receive buffer argument of the root.
47	
48	

MPI_GA	THERV(sendbuf, sendcour comm)	t, sendtype, recvbuf, recvcounts, displs, recvtype, root,	$\frac{1}{2}$
	,		3
IN	sendbuf	starting address of send buffer (choice)	4
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	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)	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	Gatherv(const void* s	endbuf, int sendcount, MPI_Datatype sendtype,	24
		const int recvcounts[], const int displs[],	25
		cvtype, int root, MPI_Comm comm)	26
MDT Co+	hory (and by f and cour	nt, sendtype, recvbuf, recvcounts, displs,	27
MF1_Gau		comm, ierror) BIND(C)	28
TYP	<pre>PE(*), DIMENSION(), 1</pre>	-	29 30
		: recvbuf	31
		sendcount, recvcounts(*), displs(*), root	32
		T(IN) :: sendtype, recvtype	33
	PE(MPI_Comm), INTENT(IN		34
	EGER, OPTIONAL, INTEN		35
MDT CAT		IT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	36
MFI_GAI	RECVTYPE, ROOT,		37
<t.v< td=""><td>pe&gt; SENDBUF(*), RECVBU</td><td></td><td>38</td></t.v<>	pe> SENDBUF(*), RECVBU		38
U	•	<pre>/PE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,</pre>	39
	M, IERROR	,, , , <u></u> , , <u>,</u> , , <u>, , , , , , , , , , , , , , , , , </u>	40
			41
		inctionality of MPI_GATHER by allowing a varying count	42
	- /	ecvcounts is now an array. It also allows more flexibility the root, by providing the new argument, displs.	43
as to wi	-	ton the sutcome is as if each process including the rest	44

If comm is an intracommunicator, the outcome is *as if* each process, including the root 45 process, sends a message to the root, 46

MPI\_Send(sendbuf, sendcount, sendtype, root, ...),

 $46 \\ 47$ 

 $\mathbf{2}$ 

3 4

5

6

7

8

9

10

11

16

32

34

 $\texttt{MPI_Recv}(\texttt{recvbuf} + \texttt{displs}[j] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[j], \texttt{recvtype}, \texttt{i}, ...).$ 

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).

and the root executes n receives,

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

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

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

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

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

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

# <sup>29</sup> 5.5.1 Examples using MPI\_GATHER, MPI\_GATHERV

<sup>31</sup> The examples in this section use intracommunicators.

### 33 Example 5.2

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

```
35
         MPI_Comm comm;
36
         int gsize,sendarray[100];
37
         int root, *rbuf;
38
          . . .
39
         MPI_Comm_size(comm, &gsize);
40
         rbuf = (int *)malloc(gsize*100*sizeof(int));
41
         MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
42
43
44
     Example 5.3
45
         Previous example modified – only the root allocates memory for the receive buffer.
46
47
```

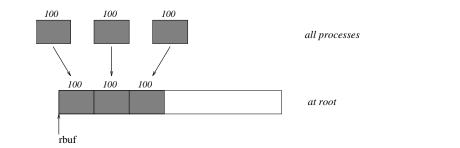


Figure 5.4: The root process gathers 100 ints from each process in the group.

```
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_conmit(&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
\mathbf{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>
```

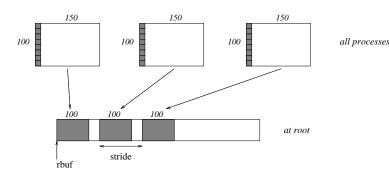


Figure 5.6: The root process gathers column 0 of a  $100 \times 150$  C array, and each set is placed stride ints apart.

```
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

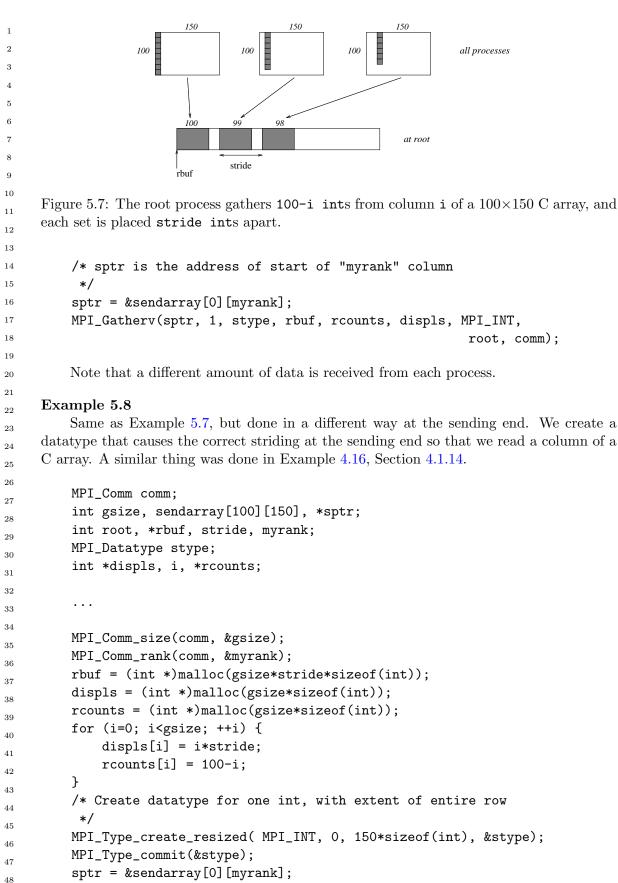
21

22 23 24

25

26

```
CHAPTER 5. COLLECTIVE COMMUNICATION
```



```
MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                           root, comm);
```

2

3 4

5

6

7 8

47

48

#### 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;
                                                                                 9
int gsize,sendarray[100][150],*sptr;
                                                                                 10
int root, *rbuf, *stride, myrank, bufsize;
                                                                                 11
MPI_Datatype stype;
                                                                                 12
int *displs,i,*rcounts,offset;
                                                                                 13
                                                                                 14
. . .
                                                                                 15
                                                                                 16
MPI_Comm_size(comm, &gsize);
                                                                                 17
MPI_Comm_rank(comm, &myrank);
                                                                                 18
                                                                                 19
stride = (int *)malloc(gsize*sizeof(int));
                                                                                 20
. . .
                                                                                 21
/* stride[i] for i = 0 to gsize-1 is set somehow
                                                                                 22
*/
                                                                                 23
                                                                                 24
/* set up displs and rcounts vectors first
                                                                                 25
 */
                                                                                 26
displs = (int *)malloc(gsize*sizeof(int));
                                                                                 27
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                 28
offset = 0;
                                                                                 29
for (i=0; i<gsize; ++i) {</pre>
                                                                                 30
    displs[i] = offset;
                                                                                 31
    offset += stride[i];
                                                                                 32
    rcounts[i] = 100-i;
                                                                                 33
}
                                                                                 34
/* the required buffer size for rbuf is now easily obtained
                                                                                 35
 */
                                                                                 36
bufsize = displs[gsize-1]+rcounts[gsize-1];
                                                                                 37
rbuf = (int *)malloc(bufsize*sizeof(int));
                                                                                 38
/* Create datatype for the column we are sending
                                                                                 39
 */
                                                                                 40
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                 41
MPI_Type_commit(&stype);
                                                                                 42
sptr = &sendarray[0][myrank];
                                                                                 43
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                                                 44
                                                        root, comm);
                                                                                 45
                                                                                 46
```

Example 5.10

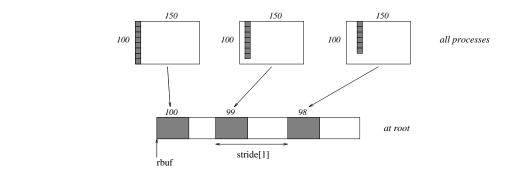


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

18

1

2

7

8

9 10

11

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

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

and each process executed a receive,

2 3 4

5

6

1

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

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

7 8 9

10

11

12

13

14

15

16

17

18 19

20

21

22

The send buffer is ignored for all non-root processes. The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps

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

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

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

<sup>23</sup> The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE as <sup>24</sup> the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and <sup>25</sup> root "sends" no data to itself. The scattered vector is still assumed to contain n segments, <sup>26</sup> where n is the group size; the *root*-th segment, which root should "send to itself," is not <sup>27</sup> moved.

<sup>28</sup> If comm is an intercommunicator, then the call involves all processes in the intercom-<sup>29</sup> municator, but with one group (group A) defining the root process. All processes in the <sup>30</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>31</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>32</sup> pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes in <sup>33</sup> group B. The receive buffer arguments of the processes in group B must be consistent with <sup>34</sup> the send buffer argument of the root.

- 47 48

<pre>NM Sendounts integration of the form of the form</pre>	IN	sendbuf	address of send buffer (choice, significant only at root)
<pre>ifying the number of elements to send to each rank ifying the number of elements to send to each rank 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) at MPI_Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) PI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(IN) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVEOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV is the inverse operation to MPI_SCATTER by allowing a varying punt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>			
<pre>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) IN comm communicator (handle) IN comm communicator (handle) IN comm communicator (handle) II scatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) II_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(IN) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	IIN	senacounts	
<pre>take the outgoing data to process i int 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) it MPI_Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) i_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows </type></pre>	IN	displs	
<pre>N sendtype data type of send buffer elements (handle) OUT recvbuf address of receive buffer (choice) N recvcount number of elements in receive buffer (non-negative in-</pre>			
OUT       recvbuf       address of receive buffer (choice)         IN       recvcount       number of elements in receive buffer (non-negative integer)         IN       recvtype       data type of receive buffer elements (handle)         IN       root       rank of sending process (integer)         IN       comm       communicator (handle)         IN       const       int sendcounts[], const         Int recvtype, cost, comm, ierror)       BIND(C)         TYPE(MPI_Datatype), INTENT(IN) ::       sendtype, recvtype         TYPE(MPI_Datatype), INTENT(IN) ::       sendtype, recvtype         TYPE(MPI_Datatype), INTENT(IN) ::       sendtype, recvtype         TYPE(MPI_Da			
<pre>N recvcount number of elements in receive buffer (non-negative in- teger) N recvtype data type of receive buffer elements (handle) N root rank of sending process (integer) N comm communicator (handle) t MPI_Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) I_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION(), is recvbuf INTEGER, INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type>SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows </type></pre>	IN	sendtype	data type of send buffer elements (handle)
<pre>teger) N recvtype data type of receive buffer elements (handle) N root rank of sending process (integer) N comm communicator (handle) t MPI_Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) I_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows </type></pre>	OUT	recvbuf	address of receive buffer (choice)
<pre>N recvtype data type of receive buffer elements (handle) N root rank of sending process (integer) N comm communicator (handle) t MPI_Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) T_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: ierror T_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVEOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDEBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	IN	recvcount	number of elements in receive buffer (non-negative in-
<pre>N root rank of sending process (integer) N comm communicator (handle) At MPI_Scatterv(const void* sendbuf, const int sendcounts[], const</pre>			teger)
<pre>N comm communicator (handle) t MPI_Scatterv(const void* sendbuf, const int sendcounts[], const</pre>	IN	recvtype	data type of receive buffer elements (handle)
<pre>At MPI_Scatterv(const void* sendbuf, const int sendcounts[], const</pre>	IN	root	rank of sending process (integer)
<pre>int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) PI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV is the inverse operation to MPI_SCATTER by allowing a varying ount of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	IN	comm	communicator (handle)
<pre>int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) I_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>			
<pre>int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) I_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,</pre>	t MPI	_Scatterv(const void	<pre>* sendbuf, const int sendcounts[], const</pre>
<pre>PI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>		<pre>int displs[],</pre>	MPI_Datatype sendtype, void* recvbuf,
<pre>recvtype, root, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>		int recvcount,	MPI_Datatype recvtype, int root, MPI_Comm comm)
<pre>TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>			
<pre>TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV is the inverse operation to MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	I_Sca	tterv(sendbuf, sendco	ounts, displs, sendtype, recvbuf, recvcount,
<pre>INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	'I_Sca		
<pre>TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	TYP	recvtype, root E(*), DIMENSION(),	, comm, ierror) BIND(C) INTENT(IN) :: sendbuf
<pre>TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror PI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type></pre>	TYP TYP	recvtype, root E(*), DIMENSION(), E(*), DIMENSION()	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf</pre>
<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,</pre>	TYP TYP INT	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) ::	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root</pre>
<pre>I_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,</pre>	TYP TYP INT TYP	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INT	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype</pre>
RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type>	TYP TYP INT TYP TYP	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT()	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm</pre>
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows</type>	TYP TYP INT TYP TYP	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT( EGER, OPTIONAL, INTE	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror</pre>
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows	TYP TYP INT TYP TYP INT	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT(1) EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDCO	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,</pre>
COMM, IERROR MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows	TYP TYP INT TYP TYP INT	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT( EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, , COMM, IERROR)</pre>
MPI_SCATTERV is the inverse operation to MPI_GATHERV. MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows	TYP TYP INT TYP INT I_SCA <ty< td=""><td>recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT() EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe&gt; SENDBUF(*), RECVI</td><td><pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*)</pre></td></ty<>	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT() EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVI	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*)</pre>
MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying unt of data to be sent to each process, since sendcounts is now an array. It also allows	TYP TYP INT TYP INT I_SCA <ty INT</ty 	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT() EGER, OPTIONAL, INTE TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVI EGER SENDCOUNTS(*), I	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*)</pre>
unt of data to be sent to each process, since sendcounts is now an array. It also allows	TYP TYP INT TYP INT I_SCA <ty INT COM</ty 	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT() EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVN EGER SENDCOUNTS(*), N M, IERROR	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,</pre>
	TYP TYP INT TYP INT I_SCA <ty INT COM</ty 	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT(: EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVI EGER SENDCOUNTS(*), I M, IERROR _SCATTERV is the inver	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, ', COMM, IERROR) BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, rse operation to MPI_GATHERV.</pre>
	TYP TYP INT TYP INT PI_SCA <ty INT COM MPI MPI</ty 	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT() EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVI EGER SENDCOUNTS(*), I M, IERROR _SCATTERV is the inver _SCATTERV extends th	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, rse operation to MPI_GATHERV. he functionality of MPI_SCATTER by allowing a varying</pre>
	TYP TYP INT TYP INT PI_SCA <ty INT COM MPI Dunt of ore flee</ty 	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT() EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVH EGER SENDCOUNTS(*), H M, IERROR _SCATTERV is the inver _SCATTERV extends the data to be sent to each	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, rse operation to MPI_GATHERV. he functionality of MPI_SCATTER by allowing a varying process, since sendcounts is now an array. It also allows</pre>
If comm is an intracommunicator, the outcome is as if the root executed n send oper-	TYP TYP INT TYP INT PI_SCA <ty INT COM MPI MPI ount of core flex</ty 	recvtype, root E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTH E(MPI_Comm), INTENT() EGER, OPTIONAL, INTEN TTERV(SENDBUF, SENDCO RECVTYPE, ROOT pe> SENDBUF(*), RECVI EGER SENDCOUNTS(*), I M, IERROR _SCATTERV is the inver _SCATTERV extends the data to be sent to each cibility as to where the o t, displs.	<pre>, comm, ierror) BIND(C) INTENT(IN) :: sendbuf :: recvbuf sendcounts(*), displs(*), recvcount, root ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, IERROR) BUF(*) DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, rse operation to MPI_GATHERV. he functionality of MPI_SCATTER by allowing a varying process, since sendcounts is now an array. It also allows lata is taken from on the root, by providing an additional</pre>

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

```
\texttt{MPI}\_\texttt{Send}(\texttt{sendbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...), \qquad {}^{45}_{46}
```

and each process executed a receive,

```
\texttt{MPI\_Recv}(\texttt{recvbuf},\texttt{recvcount},\texttt{recvtype},\texttt{i},\ldots).
```

# Unofficial Draft for Comment Only

44

47

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 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>10</sup> The specification of counts, types, and displacements should not cause any location on <sup>11</sup> the root to be read more than once.

<sup>12</sup> The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE as <sup>13</sup> the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and <sup>14</sup> root "sends" no data to itself. The scattered vector is still assumed to contain n segments, <sup>15</sup> where n is the group size; the *root*-th segment, which root should "send to itself," is not <sup>16</sup> moved.

<sup>17</sup> If comm is an intercommunicator, then the call involves all processes in the intercom-<sup>18</sup> municator, but with one group (group A) defining the root process. All processes in the <sup>19</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>20</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>21</sup> pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes in <sup>22</sup> group B. The receive buffer arguments of the processes in group B must be consistent with <sup>23</sup> the send buffer argument of the root.

 $\frac{24}{25}$ 

26

32

33

34

35

36

37

38 39

40 41

47 48

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

<sup>27</sup> The examples in this section use intracommunicators.

#### <sup>28</sup> <sub>29</sub> Example 5.11

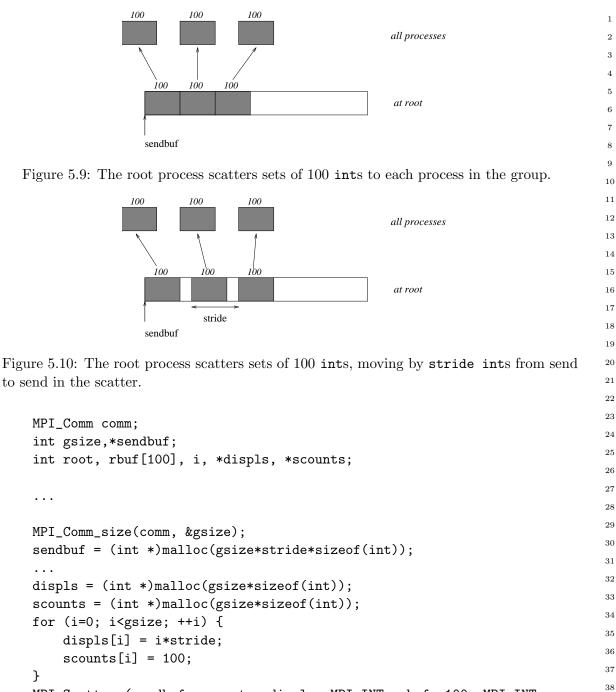
The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9.

MPI\_Comm comm; int gsize,\*sendbuf; int root, rbuf[100]; ... MPI\_Comm\_size(comm, &gsize); sendbuf = (int \*)malloc(gsize\*100\*sizeof(int)); ... MPI\_Scatter(sendbuf, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm);

# <sup>42</sup> Example 5.12

<sup>43</sup> The reverse of Example 5.5. The root process scatters sets of 100 ints to the other <sup>44</sup> processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of <sup>45</sup> MPI\_SCATTERV. Assume *stride*  $\geq$  100. See Figure 5.10.

Unofficial Draft for Comment Only



```
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) {</pre>
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
```

```
root, comm);
```

40 41

42

43

44

4546

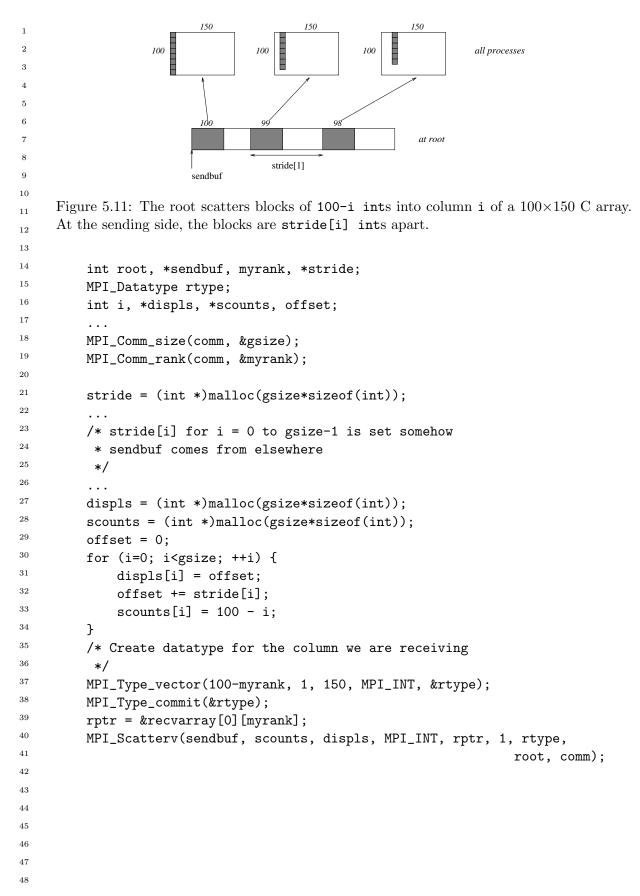
47

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

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)	1
OUT	recvbuf	address of receive buffer (choice)	1
IN	recvcount	number of elements received from any process (non-negative integer)	1
IN	recvtype	data type of receive buffer elements (handle)	1
IN	comm	communicator (handle)	1
			1
int MPI	-	d* sendbuf, int sendcount,	1
	• •	endtype, void* recvbuf, int recvcount,	1
	MPI_Datatype r	ecvtype, MPI_Comm comm)	1
MPI_All	•	count, sendtype, recvbuf, recvcount, recvtype,	
	comm, ierror)		1
		INTENT(IN) :: sendbuf	:
	E(*), DIMENSION()		:
	EGER, INTENT(IN) ::		1
	E(MPI_Datatype), INI E(MPI_Comm), INTENT(	ENT(IN) :: sendtype, recvtype	1
	EGER, OPTIONAL, INTE		1
MPI_ALL		COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	
	COMM, IERROR)		
•	pe> SENDBUF(*), RECV	TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	
		ought of as MPI_GATHER, but where all processes receive	
		t. 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	
	-	th recvcount, recvtype at any other process. cator, the outcome of a call to MPI_ALLGATHER() is as	
	cesses executed n calls t		
n an pro	Cosses executed in calls t	U	
MPI_	Gather(sendbuf,sendc	ount,sendtype,recvbuf,recvcount,	
		recvtype,root,comm)	
for root	= 0 ,, n-1. Then	rules for correct usage of MPI_ALLGATHER are easily found	
	corresponding rules for		
The	"in place" option for in	tracommunicators is specified by passing the value	

MPI\_IN\_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored.

# Unofficial Draft for Comment Only

1

Then the input data of each process is assumed to be in the area where that process would
 receive its own contribution to the receive buffer.
 If comm is an intercommunicator, then each process of one group (group A) contributes

sendcount data items; these data are concatenated and the result is stored at each process
 in the other group (group B). Conversely the concatenation of the contributions of the
 processes in group B is stored at each process in group A. The send buffer arguments in
 group A must be consistent with the receive buffer arguments in group B, and vice versa.

Advice to users. The communication pattern of MPI\_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount = 0 for the communication in the reverse direction.

- (End of advice to users.)
- 18 19 20

21 22

47

48

8

9

10

11

12

13

14

15

16 17

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

23IN sendbuf starting address of send buffer (choice)  $^{24}$ IN sendcount number of elements in send buffer (non-negative inte-25ger) 26data type of send buffer elements (handle) IN sendtype 2728OUT recvbuf address of receive buffer (choice) 29 IN non-negative integer array (of length group size) conrecvcounts 30 taining the number of elements that are received from  $^{31}$ each process 32 IN displs integer array (of length group size). Entry i specifies 33 the displacement (relative to recvbuf) at which to place 34 the incoming data from process i 35 36 IN data type of receive buffer elements (handle) recvtype 37 IN comm communicator (handle) 38 39 int MPI\_Allgatherv(const void\* sendbuf, int sendcount, 40MPI\_Datatype sendtype, void\* recvbuf, const int recvcounts[], 41 const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm) 4243 MPI\_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 44recvtype, comm, ierror) BIND(C) 45TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 46

TYPE(\*), DIMENSION(..) :: recvbuf

```
INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
```

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

```
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
IERROR
```

MPI\_ALLGATHERV can be thought of as MPI\_GATHERV, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. These blocks need not all be the same size.

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

If comm is an intracommunicator, the outcome is as if all processes executed calls to

## MPI\_Gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs, recvtype,root,comm),

for root = 0 , ..., n-1. The rules for correct usage of MPI\_ALLGATHERV are easily found from the corresponding rules for MPI\_GATHERV.

The "in place" option for intracommunicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at all processes. In such a case, sendcount and sendtype are ignored, and the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (group B). Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

## 5.7.1 Example using MPI\_ALLGATHER

The example in this section uses intracommunicators.

#### Example 5.14

The all-gather version of Example 5.2. Using MPI\_ALLGATHER, we will gather 100 ints from every process in the group to every process.

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

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

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

5

CHAPTER 5. COLLECTIVE COMMUNICATION

# 5.8 All-to-All Scatter/Gather

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

IN       sendbuf       starting address of send buffer (choice)         IN       sendcount       number of elements sent to each process (non-negative integer)         IN       sendtype       data type of send buffer (choice)         IN       recvbuf       address of receive buffer (choice)         IN       recvbuf       number of elements received from any process (non-negative integer)         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)         IN       comm       communicator (handle)         IN       comm       communicator (bandle)         IN       comm       communicator (handle)         IN       comm       communicator (bandle)         IN       recvtype       data type of recvtyf, recvcount, recvtype, comm, ierror) BIDD(C)         TYPE(*), DIMENSION(), INTENT(IN) :: sendtype, recvtype       TYPE(VPI_Comm,) INTENT(IN) :: sendtype, recvtype         TYPE(VPI_Comm), INTENT(IN) :: sendtype, recvtype       TYPE(VPI_Comm,), INTENT(IN) :: sendtype, recvtype         TYPE(VPI_Comm,), INTENT(IN) :: sendtype, recvtype       COMM, IERROR) <type> SENDBUF(*), RECVEUF(*)         INTEGER, OPTIONAL, INTENT(IN) ::</type>	5			atype, recobar, recocoant, recotype, commy
IN         Sendcount         number of elements sent to each process (non-negative integer)           IN         sendtype         data type of send buffer elements (handle)           OUT         recvbuf         address of receive buffer (choice)           IN         recvtount         number of elements received from any process (non-negative integer)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (handle)           IN         recvtope         data type of receive buffer elements (battype)           INTEGER         number of elements centrype         datty	6	IN	sendbuf	starting address of send buffer (choice)
IN         sendtype         data type of send buffer elements (handle)           11         OUT         recvbuf         address of receive buffer (choice)           12         IN         recvcount         number of elements receive duffer dements (handle)           13         IN         recvtype         data type of receive buffer elements (handle)           14         IN         recvtype         data type of receive buffer elements (handle)           15         IN         recvtype         data type of receive buffer elements (handle)           16         IN         comm         communicator (handle)           17         void* recvbuf, int recvcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype sendtype, methods           18         int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, comm, ierror) BIND(C)           19         TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf           117         TYPE(*), DIMENSION() :: recvbuf           118         INTEGER, INTENT(IN) :: sendcount, recvount           119         INTEGER, OPTIONAL, INTENT(IN) :: sendtype, recvtype           119         TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype, COMM, IERROR           111         COM, IERROR           112         COM, IERROR           1111         TREGER SENDCOUNT, SENDTUPE, RECVEUU	8	IN	sendcount	
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)         IN       comm       mPI_Comm         INTEGER, Alltoall (const void* sendbupt, int recvcount, MPI_Datatype sendtype, comm, ierror) BIND(C)       TYPE(*), DIMENSION() (INTENT(IN) :: sendbuff         TYPE(*), DIMENSION()       INTENT(IN) :: sendtype, recvtype       TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype         TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype       TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype, comm, iERROR         MPI_ALLTOALL (SENDBUF, SENDCOUNT, SENDTYPE, RECVEUF, RECVCOUNT, RECVTYPE, COMM, IERROR       MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process and distinct data to each of the receivers. The j-th block sent from process is received by process j and is plac		IN	sendtype	data type of send buffer elements (handle)
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)         IN       comm       communicator (handle)         IN       comm       communicator (handle)         IN       comm       communicator (handle)         IN       mPI_Alltoall (const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, Comm, ierror) BIND(C)         INTEQER, INTENT(IN):       sendcount, recvcount         TYPE(*), DIMENSION(), INTENT(IN)::       sendbuff         INTEGER, INTENT(IN):       sendcount, recvcount         TYPE(MPI_Comm), INTENT(IN)::       comm         INTEGER, OPTIONAL, INTENT(OUT):::       ierror         MPI_ALLTOALL (SENDBUF, SENDCOUNT, SENDTYPE, RECVEUF, RECVCOUNT, RECVTYPE, COMM, IERROR         Cuppe> SENDEUF(*), RECVEUF(*)         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR         MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process         sends distinct data to each of the receivers. The j-th block sent from process i is received         by process j and is placed in the i-th block of recvbuf.         The type signature associated with sendcount, sendtype, at a process must be equal to	11	OUT	recvbuf	address of receive buffer (choice)
1N       recvtype       data type of receive buffer elements (handle)         16       IN       comm       communicator (handle)         17       int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_COmm comm)         18       int MPI_Alltoall(const void* sendbuf, int recvcount, MPI_Datatype recvtype, comm, ierror) BIND(C)         11       TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf         12       TYPE(*), DIMENSION() :: recvbuf         13       INTEGER, INTENT(IN) :: sendtount, recvtype, recvtype         14       INTEGER, INTENT(IN) :: sendtype, recvtype         15       TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype         16       INTEGER, OPTIONAL, INTENT(OUT) :: ierror         17       MPI_ALLTOALL (SENDBUF, SENDCOUNT, SENDTYPE, RECVEUF, RECVCOUNT, RECVTYPE, COMM, IERROR)         16       COMM, IERROR)         17       COMM, IERROR         18       MPI_ALLTOALL (SENDBUF(*), RECVBUF(*)         18       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR         18       MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.         18       The type signature associated with sendcount, sendtype, at a process must be equal	13	IN	recvcount	• <u>-</u> (
10       comm       communicator (handle)         17       int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)         18       int MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) BIND(C)         19       MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) BIND(C)         14       TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf         15       TYPE(*), DIMENSION() :: recvbuf         16       INTEGER, INTENT(IN) :: sendcount, recvcount         17       TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype         18       TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtupe         19       INTEGER, OPTIONAL, INTENT(OUT) :: ierror         10       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR         10       COMM, IERROR)         24       COMM, IERROR)         25       sends distinct data to each of the receivers. The j-th block sent from process i is received         26       by process j and is placed in the i-th block of recvbuf.         27       The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with sendcount, sendtype, at a process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be differe		IN	recvtype	data type of receive buffer elements (handle)
<pre>int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,</pre>	16	IN	comm	communicator (handle)
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) (type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different. If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to, MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i,), and a receive from every other process with a call to, MPI_Bacw(receivbif + i · recureount · extent(sendtype), recureount recuture i)	18 19 20	int MPI_Al	<pre>void* recvbuf, int re</pre>	VI VI
<ul> <li>MPI_ALLIVALL(SEADBOF, SEADCOONT, SEADCOONT, SEADTIPE, RECVEDINT, RECVERTIPE, COMM, IERROR</li> <li>MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.</li> <li>The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.</li> <li>If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,</li> <li>MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i,), and a receive from every other process with a call to,</li> </ul>	22 23 24 25 26 27 28	TYPE(* TYPE(* INTEGE TYPE(M TYPE(M	<pre>comm, ierror) BIND(C) (), DIMENSION(), INTENT (), DIMENSION() :: re (), INTENT(IN) :: sendco () PI_Datatype), INTENT(IN) () PI_Comm), INTENT(IN) ::</pre>	) C(IN) :: sendbuf ecvbuf punt, recvcount 0 :: sendtype, recvtype comm
<ul> <li>each process (itself included) with a call to,</li> <li>MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i,),</li> <li>and a receive from every other process with a call to,</li> <li>MPI_Back(recybuf + i · recycount · extent(recytype) recycount recytype i)</li> </ul>	31 32 33 34 35 36 37 38 39 40	<type> INTEGE MPI_A sends distin by process ; The typ the type sig that the am</type>	COMM, IERROR) SENDBUF(*), RECVBUF(*) CR SENDCOUNT, SENDTYPE, F LLTOALL is an extension of ct data to each of the receive j and is placed in the i-th bl pe signature associated with gnature associated with recvo ount of data sent must be equ	RECVCOUNT, RECVTYPE, COMM, IERROR MPI_ALLGATHER to the case where each process ers. The j-th block sent from process i is received ock of recvbuf. 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
<sup>45</sup> MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i,), <sup>46</sup> and a receive from every other process with a call to, <sup>47</sup> MPI_Back(recybuf + i · recycount · extent(recytype) recycount recytype i)				_
47 MPI $\text{Recu(recubuf} \pm i, recucount, extent(recutupe) recucount, recutupe i))$		MPI_S	$\texttt{end}(\texttt{sendbuf}+\texttt{i}\cdot\texttt{sendcount})$	$\texttt{t} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i},),$
MPT $\text{Recu(recubuf} \pm i, recucount, extent(recuture))$ recucount recuture i	46	and a receiv	ve from every other process w	with a call to,
		MPI_R	$\texttt{ecv}(\texttt{recvbuf}+\texttt{i}\cdot\texttt{recvcount})$	$t \cdot extent(recvtype), recvcount, recvtype, i,).$

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

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

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

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

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

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

(End of advice to users.)

 $\mathbf{2}$ 

1 2	MPI_ALLT(	DALLV(sendbuf, sendcounts, se recvtype, comm)	displs, sendtype, recvbuf, recvcounts, rdispls,
3 4	IN	sendbuf	starting address of send buffer (choice)
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 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 35	TYPE(* TYPE(* INTEGH rdisp] TYPE()	<pre>int recvcounts[], co MPI_Comm comm) allv(sendbuf, sendcounts rdispls, recvtype, c *), DIMENSION(), INTENT *), DIMENSION() :: re ER, INTENT(IN) :: sendco</pre>	<pre>F(IN) :: sendbuf ecvbuf ounts(*), sdispls(*), recvcounts(*), ) :: sendtype, recvtype</pre>
36		ER, OPTIONAL, INTENT(OUT)	
37 38 39 40 41 42	<type> INTEGH</type>	RDISPLS, RECVTYPE, C > SENDBUF(*), RECVBUF(*)	, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, OMM, IERROR) S(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
43 44 45 46 47 48	the send is side is speci If com	specified by sdispls and the la ified by rdispls. m is an intracommunicator, t	b MPI_ALLTOALL in that the location of data for boation of the placement of the data on the receive then the j-th block sent from process i is received lock of recvbuf. These blocks need not all have the

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,

 $MPI\_Send(sendbuf + sdispls[i] \cdot extent(sendtype), sendcounts[i], sendtype, i, ...),$ 

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.)

 $^{24}$ 

12	MPI_ALLT	OALLW(sendbuf, sendcounts, s recvtypes, comm)	sdispls, sendtypes, recvbuf, recvcounts, rdispls,
3	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 14	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
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 specifies the type of data received from process i (array of handles)
27	IN	comm	communicator (handle)
28 29 30 31 32 33	int MPI_A	<pre>int sdispls[], const</pre>	<pre>dbuf, const int sendcounts[], const     MPI_Datatype sendtypes[], void* recvbuf, [], const int rdispls[], const es[], MPI_Comm comm)</pre>
34 35 36 37 38 39 40 41 42 43 44	TYPE( TYPE( INTEG rdisp TYPE( TYPE( INTEG	rdispls, recvtypes, (*), DIMENSION(), INTEN (*), DIMENSION() :: re (ER, INTENT(IN) :: sendco ols(*) (MPI_Datatype), INTENT(IN) (MPI_Datatype), INTENT(IN) (MPI_Comm), INTENT(IN) :: (ER, OPTIONAL, INTENT(OUT)	<pre>ecvbuf ounts(*), sdispls(*), recvcounts(*), ) :: sendtypes(*) ) :: recvtypes(*) comm</pre>
45 46 47 48	<type INTEG</type 	RDISPLS, RECVTYPES, >> SENDBUF(*), RECVBUF(*)	COMM, IERROR) S(*), 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.

 $\mathbf{2}$ 

 $^{24}$ 

```
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
                   count
                                                number of elements in send buffer (non-negative inte-
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
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
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
44
      \operatorname{recvbuf}(1) = \operatorname{global}\max(\operatorname{sendbuf}(1)) and \operatorname{recvbuf}(2) = \operatorname{global}\max(\operatorname{sendbuf}(2)).
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 45 46 46 47 88

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23 24

25

26

27

28

29

30

31

32

33 34

35

36

37

38

39

40 41

	176	CHAPTER 5. COLLECTIVE COMMUNICATION
1	5.9.2 Predefined Reduction Operat	ions
2 3 4 5 6 7	MPI_ALLREDUCE, MPI_REDUCE_S( MPI_SCAN, MPI_EXSCAN, all nonb	are supplied for MPI_REDUCE and related functions CATTER_BLOCK, MPI_REDUCE_SCATTER, blocking 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	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 Sec- operations, we enumerate below the allowed combi- s. First, define groups of MPI basic datatypes in the</pre>
25 26 27	following way.	5. Thise, define groups of with basic datatypes in the
28 29 30 31 32 33 34 35 36 37 38	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,
39 40 41 42 43 44	Fortran integer:	MPI_UINT32_T, MPI_UINT64_T MPI_INTEGER, and handles returned from MPI_TYPE_CREATE_F90_INTEGER, and if available: MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, MPI_INTEGER16
45 46 47 48	Floating point:	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
	C	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
		, , ,	19
	Now, the valid datatypes for each open	ration are specified below.	20
			21
			22
	Ор	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 operations together with all liste	ed datatypes are valid in all supported program-	31
n	ning languages, see also Reduce Operation		32
	The following examples use intracomm		33
	The following examples use moracolini		34
E	Example 5.15		35
	-	luct of two vectors that are distributed across a	36
o	roup of processes and returns the answer		
8	To ap of processes and retains the answer		37
			38
			39
			40
			41
			42
			43
			44
			45
			46
			47
			48

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

 $^{47}$  ters) cannot be used in reduction operations. In a heterogeneous environment, MPI\_CHAR,

<sup>48</sup> 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 

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

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

Each process has an array of 30 doubles, in C. For each of the 30 locations, compute
 the value and rank of the process containing the largest value.

```
1
    . . .
                                                                                       \mathbf{2}
    /* each process has an array of 30 double: ain[30]
                                                                                       3
     */
                                                                                       4
    double ain[30], aout[30];
    int ind[30];
                                                                                       5
                                                                                       6
    struct {
                                                                                       7
        double val;
                                                                                        8
              rank;
        int
    } in[30], out[30];
                                                                                       9
                                                                                       10
    int i, myrank, root;
                                                                                       11
    MPI_Comm_rank(comm, &myrank);
                                                                                       12
    for (i=0; i<30; ++i) {
                                                                                       13
                                                                                       14
        in[i].val = ain[i];
                                                                                       15
        in[i].rank = myrank;
                                                                                       16
    }
                                                                                       17
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                       18
    /* At this point, the answer resides on process root
                                                                                       19
     */
                                                                                       20
    if (myrank == root) {
                                                                                       21
        /* read ranks out
         */
                                                                                       22
        for (i=0; i<30; ++i) {</pre>
                                                                                       23
                                                                                       ^{24}
             aout[i] = out[i].val;
                                                                                       25
             ind[i] = out[i].rank;
                                                                                       26
        }
    }
                                                                                       27
                                                                                       28
                                                                                       29
Example 5.18
                                                                                       30
   Same example, in Fortran.
                                                                                       ^{31}
                                                                                       32
                                                                                       33
    ! each process has an array of 30 double: ain(30)
                                                                                       34
                                                                                       35
    DOUBLE PRECISION ain(30), aout(30)
                                                                                       36
    INTEGER ind(30)
                                                                                       37
    DOUBLE PRECISION in(2,30), out(2,30)
                                                                                       38
    INTEGER i, myrank, root, ierr
                                                                                       39
                                                                                       40
    CALL MPI_COMM_RANK(comm, myrank, ierr)
                                                                                       41
    DO I=1, 30
                                                                                       42
        in(1,i) = ain(i)
                                                                                       43
        in(2,i) = myrank ! myrank is coerced to a double
                                                                                       44
    END DO
                                                                                       45
                                                                                       46
    CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                                                       47
                                                                     comm, ierr)
                                                                                       48
```

```
1
          ! At this point, the answer resides on process root
\mathbf{2}
3
         IF (myrank .EQ. root) THEN
4
              ! read ranks out
5
              DO I= 1, 30
6
                   aout(i) = out(1,i)
7
                   ind(i) = out(2,i) ! rank is coerced back to an integer
8
              END DO
9
         END IF
10
11
     Example 5.19
12
         Each process has a non-empty array of values. Find the minimum global value, the
13
     rank of the process that holds it and its index on this process.
14
15
     #define LEN
                      1000
16
17
     float val[LEN];
                               /* local array of values */
^{18}
                               /* local number of values */
     int count;
19
     int myrank, minrank, minindex;
20
     float minval;
21
22
     struct {
23
         float value;
^{24}
         int
                index;
25
     } in, out;
26
27
         /* local minloc */
28
     in.value = val[0];
     in.index = 0;
29
30
     for (i=1; i < count; i++)</pre>
^{31}
          if (in.value > val[i]) {
32
              in.value = val[i];
33
              in.index = i;
34
         }
35
36
         /* global minloc */
37
     MPI_Comm_rank(comm, &myrank);
38
     in.index = myrank*LEN + in.index;
39
     MPI_Reduce( &in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
40
          /* At this point, the answer resides on process root
41
           */
42
     if (myrank == root) {
43
         /* read answer out
44
           */
45
         minval = out.value;
46
         minrank = out.index / LEN;
47
         minindex = out.index % LEN;
48
     }
```

*Rationale.* The definition of MPI\_MINLOC and MPI\_MAXLOC given here has the 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)

IN	user_fn	user defined function (function)
IN	commute	<b>true</b> if commutative; <b>false</b> otherwise.
OUT	ор	operation (handle)

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 userdefined 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. 41
typedef void MPI_User_function(void* invec, void* inoutvec, int *len,
MPI_Datatype *datatype); 43
The Fortran declarations of the user-defined function user_fn appear below. 45
ABSTRACT INTERFACE 46
```

SUBROUTINE MPI\_User\_function(invec, inoutvec, len, datatype) BIND(C) USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR  $^{24}$ 

1	
1 2	TYPE(C_PTR), VALUE :: invec, inoutvec
3	INTEGER :: len
4	TYPE(MPI_Datatype) :: datatype
4 5	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
	<type> INVEC(LEN), INOUTVEC(LEN)</type>
6	INTEGER LEN, DATATYPE
7	
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 $v[0],, v[len-1]$ 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	$i = 0, \dots, count - 1$ , where $\circ$ is the combining operation computed by the function.
22 23	Rationale. The len argument allows MPI_REDUCE to avoid calling the function for
23 24	each element in the input buffer. Rather, the system can choose to apply the function
24 25	to chunks of input. In C, it is passed in as a reference for reasons of compatibility
25 26	with Fortran.
20	
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
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	may be called inside the function in case of an error.
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 datatype argument that it is passed, and cannot identify, by itself,
40	the correspondence between the datatype handles and the datatype they represent.
41	This correspondence was established when the datatypes were created. Before the
42	library is used, a library initialization preamble must be executed. This preamble
43	code will define the datatypes that are used by the library, and store handles to these
44	datatypes in global, static variables that are shared by the user code and the library
45	code.
46	The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using
47	the Fortran calling conventions and will pass a Fortran-type datatype argument; the
48	C version will use C calling convention and the C representation of a datatype handle.
	• version will use • canning convention and the • representation of a datatype fiandle.

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.*)

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

```
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 +
29
                           inout->imag*in->real;
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)

-			35
IN	sendbuf	starting address of send buffer (choice)	36
OUT	recvbuf	starting address of receive buffer (choice)	37
IN	count	number of elements in send buffer (non-negative integer)	38 39
IN	datatype	data type of elements of send buffer (handle)	40 $41$
IN	ор	operation (handle)	42
IN	comm	communicator (handle)	43
			44

int MPI\_Allreduce(const void\* sendbuf, void\* recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)

MPI\_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)

 $\mathbf{2}$ 

 $^{31}$ 

1	
2	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf
	TYPE(*), DIMENSION() :: recvbuf
3	INTEGER, INTENT(IN) :: count
4	TYPE(MPI_Datatype), INTENT(IN) :: datatype
5	TYPE(MPI_Op), INTENT(IN) :: op
6	TYPE(MPI_Comm), INTENT(IN) :: comm
7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8	MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
9	<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>
10	INTEGER COUNT, DATATYPE, OP, COMM, IERROR
11	INTEGEN COUNT, DATATITE, OF, COMM, TENNON
12	If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as
13	MPI_REDUCE except that the result appears in the receive buffer of all the group members.
14	
15	Advice to implementors. The all-reduce operations can be implemented as a re-
16	duce, followed by a broadcast. However, a direct implementation can lead to better
17	performance. (End of advice to implementors.)
18	
19	The "in place" option for intracommunicators is specified by passing the value
20	MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is
21	taken at each process from the receive buffer, where it will be replaced by the output data.
22	If comm is an intercommunicator, then the result of the reduction of the data provided
23	by processes in group A is stored at each process in group B, and vice versa. Both groups
24	should provide <b>count</b> and <b>datatype</b> arguments that specify the same type signature.
25	The following example uses an intracommunicator.
26	
27	Example 5.22
28	A routine that computes the product of a vector and an array that are distributed
29	across a group of processes and returns the answer at all nodes (see also Example $5.16$ ).
30	
31	SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
32	REAL a(m), b(m,n) ! local slice of array
33	REAL c(n) ! result
34	REAL sum(n)
35	INTEGER n, comm, i, j, ierr
36	
37	! local sum
38	DO j= 1, n
39	sum(j) = 0.0
40	DO i = 1, m
41	sum(j) = sum(j) + a(i)*b(i,j)
42	END DO
43	END DO
44	
45	! global sum
46	CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
47	
48	! return result at all nodes

RETURN					
END	END				
	3				
5.9.7 Pro	ocess-Local Reduction		4		
The functi	ons in this section are of imp	ortance to library implementors who may want to	5 6		
	_	at are otherwise not easily covered by the standard	7		
MPI opera			8		
The fo	The following function applies a reduction operator to local arguments.				
			10		
MPI_REDU	JCE_LOCAL( inbuf, inoutbuf, o	count, datatype, op)	11 12		
IN	inbuf	input buffer (choice)	12		
INOUT	inoutbuf	combined input and output buffer (choice)	14		
IN	count	number of elements in inbuf and inoutbuf buffers (non-	15		
11 1	count	negative integer)	16		
IN	datatype	data type of elements of inbuf and inoutbuf buffers	17 18		
	uatatype	(handle)	18		
IN	ор	operation (handle)	20		
	ор	oporation (namalo)	21		
int MPI_R	educe_local(const void* :	inbuf, void* inoutbuf, int count,	22		
_	MPI_Datatype datatyp		23		
MPT Reduc	e local(inbuf inoutbuf	count datatype op jerror) BIND(C)	24 25		
	<pre>MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C) 2 TYPE(*), DIMENSION(), INTENT(IN) :: inbuf 2</pre>				
		noutbuf	27		
	ER, INTENT(IN) :: count		28		
	MPI_Datatype), INTENT(IN)		29		
	<pre>MPI_Op), INTENT(IN) ::  er, OPTIONAL, INTENT(OUT)</pre>	-	30		
			31 32		
		COUNT, DATATYPE, OP, IERROR)	33		
• 1	> INBUF(*), INOUTBUF(*) ER COUNT, DATATYPE, OP, I	TEDDUD	34		
			35		
		given by op element-wise to the elements of inbuf	36		
		nent-wise in inoutbuf, as explained for user-defined f and inoutbuf (input as well as result) have the	37		
-		and moutour (input as wen as result) have the int and the same datatype given by datatype. The	38 39		
	ACE option is not allowed.	it and the same addaugpe given sy <b>detacype</b> . The	40		
	etion operations can be querie	d for their commutativity.	41		
			42		
MPI OP (	COMMUTATIVE( op, commute		43		
IN IN		operation (handle)	44 45		
	op		45 46		
OUT	commute	<b>true</b> if <b>op</b> is commutative, <b>false</b> otherwise (logical)	47		
	48				

```
1
     int MPI_Op_commutative(MPI_Op op, int *commute)
\mathbf{2}
     MPI_Op_commutative(op, commute, ierror) BIND(C)
3
          TYPE(MPI_Op), INTENT(IN) :: op
4
          LOGICAL, INTENT(OUT) :: commute
5
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
6
\overline{7}
     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
8
          LOGICAL COMMUTE
9
          INTEGER OP, IERROR
10
11
             Reduce-Scatter
12
     5.10
13
14
     MPI includes variants of the reduce operations where the result is scattered to all processes
     in a group on return. One variant scatters equal-sized blocks to all processes, while another
15
16
     variant scatters blocks that may vary in size for each process.
17
18
     5.10.1 MPI_REDUCE_SCATTER_BLOCK
19
20
21
     MPI_REDUCE_SCATTER_BLOCK( sendbuf, recvbuf, recvcount, datatype, op, comm)
22
       IN
                 sendbuf
                                             starting address of send buffer (choice)
23
^{24}
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
25
       IN
                 recvcount
                                             element count per block (non-negative integer)
26
       IN
                 datatype
                                             data type of elements of send and receive buffers (han-
27
                                             dle)
28
29
       IN
                                             operation (handle)
                 ор
30
       IN
                                             communicator (handle)
                 comm
^{31}
32
     int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,
33
                    int recvcount, MPI_Datatype datatype, MPI_Op op,
34
                    MPI_Comm comm)
35
36
     MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
37
                    ierror) BIND(C)
38
          TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                      sendbuf
39
          TYPE(*), DIMENSION(..) :: recvbuf
40
          INTEGER, INTENT(IN) :: recvcount
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_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
46
                    IERROR)
47
          <type> SENDBUF(*), RECVBUF(*)
48
```

INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR

If comm is an intracommunicator, MPI\_REDUCE\_SCATTER\_BLOCK first performs a global, element-wise reduction on vectors of count =  $n^{*}$ recvcount elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcount, datatype, op and comm. The resulting vector is treated as n consecutive blocks of recvcount elements that are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcount, and datatype.

Advice to implementors. The MPI\_REDUCE\_SCATTER\_BLOCK routine is functionally equivalent to: an MPI\_REDUCE collective operation with count equal to recvcount\*n, followed by an MPI\_SCATTER with sendcount equal to recvcount. However, a direct implementation may run faster. (*End of advice to implementors.*)

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.

 $\mathbf{2}$ 

 $^{24}$ 

```
1
      MPI_REDUCE_SCATTER( sendbuf, recvbuf, recvcounts, datatype, op, comm)
2
       IN
                 sendbuf
                                              starting address of send buffer (choice)
3
       OUT
                 recvbuf
                                              starting address of receive buffer (choice)
4
5
       IN
                  recvcounts
                                              non-negative integer array (of length group size) spec-
6
                                              ifying the number of elements of the result distributed
7
                                              to each process.
8
       IN
                                              data type of elements of send and receive buffers (han-
                 datatype
9
                                              dle)
10
       IN
                 op
                                              operation (handle)
11
       IN
                                              communicator (handle)
                 comm
12
13
14
      int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const
15
                     int recvcounts[], MPI_Datatype datatype, MPI_Op op,
16
                     MPI_Comm comm)
17
      MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
18
                     ierror) BIND(C)
19
          TYPE(*), DIMENSION(...), INTENT(IN) ::
                                                        sendbuf
20
          TYPE(*), DIMENSION(..) :: recvbuf
21
          INTEGER, INTENT(IN) :: recvcounts(*)
22
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
          TYPE(MPI_Op), INTENT(IN) :: op
24
          TYPE(MPI_Comm), INTENT(IN) :: comm
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
28
                     IERROR)
29
          <type> SENDBUF(*), RECVBUF(*)
30
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
31
          If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global,
32
     element-wise reduction on vectors of count = \sum_{i=0}^{n-1} recvcounts[i] elements in the send buffers
33
      defined by sendbuf, count and datatype, using the operation op, where n is the number of
34
      processes in the group of comm. The routine is called by all group members using the
35
      same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as
36
      n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The
37
      blocks are scattered to the processes of the group. The i-th block is sent to process i and
38
      stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.
39
40
           Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equiv-
41
           alent to: an MPI_REDUCE collective operation with count equal to the sum of
42
           recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. How-
43
           ever, a direct implementation may run faster. (End of advice to implementors.)
44
45
          The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in
     the sendbuf argument. In this case, the input data is taken from the receive buffer. It is
46
47
      not required to specify the "in place" option on all processes, since the processes for which
48
      recvcounts[i]==0 may not have allocated a receive buffer.
```

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same recvcounts argument, and provide input vectors of count =  $\sum_{i=0}^{n-1} \text{recvcounts}[i]$  elements stored in the send buffers, where n is the size of the group. The resulting vector from the other group is scattered in blocks of recvcounts[i] elements among the processes in the group. The number of elements count must be the same for the two groups.

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

# 5.11 Scan

5.11.1 Inclusive Scan

### MPI\_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)	20	
OUT	recvbuf	starting address of receive buffer (choice)	21	
IN	count	number of elements in input buffer (non-negative in-	22	
	count		23	
		teger)	24	
IN	datatype	data type of elements of input buffer (handle)	25	
IN	ор	operation (handle)	26	
IN	comm	communicator (handle)	27	
			28	

MPI_Scan(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
	40
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	41
<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR</type>	42
INTEGER COUNT, DATATIFE, OF, COMM, TERROR	43

If comm is an intracommunicator, MPI\_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,...,i (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 48

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{6}$ 

 $\overline{7}$ 

```
1
     for MPI_REDUCE. The type of operations supported, their semantics, and the constraints
\mathbf{2}
     on send and receive buffers are as for MPI_REDUCE.
3
          The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in
4
     the sendbuf argument. In this case, the input data is taken from the receive buffer, and
5
      replaced by the output data.
6
          This operation is invalid for intercommunicators.
7
8
     5.11.2 Exclusive Scan
9
10
11
      MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)
12
                 sendbuf
       IN
                                              starting address of send buffer (choice)
13
14
       OUT
                                              starting address of receive buffer (choice)
                 recvbuf
15
       IN
                 count
                                              number of elements in input buffer (non-negative in-
16
                                               teger)
17
       IN
                 datatype
                                              data type of elements of input buffer (handle)
18
19
       IN
                                              operation (handle)
                 op
20
       IN
                 comm
                                              intracommunicator (handle)
21
22
      int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
23
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
^{24}
25
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
26
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
27
          TYPE(*), DIMENSION(..) :: recvbuf
28
          INTEGER, INTENT(IN) :: count
29
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
          TYPE(MPI_Op), INTENT(IN) :: op
^{31}
          TYPE(MPI_Comm), INTENT(IN) :: comm
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
      MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
34
          <type> SENDBUF(*), RECVBUF(*)
35
          INTEGER COUNT, DATATYPE, OP, COMM, IERROR
36
37
          If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction
38
      on data distributed across the group. The value in recvbuf on the process with rank 0 is
39
      undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process
40
      with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes
41
      with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the
42
      reduction of the values in the send buffers of processes with ranks 0, \ldots, i-1 (inclusive). The
43
      routine is called by all group members using the same arguments for count, datatype, op and
44
      comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE.
45
      The type of operations supported, their semantics, and the constraints on send and receive
46
     buffers, are as for MPI_REDUCE.
47
48
```

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,

$$w = \left\{ egin{array}{cc} u+v & ext{if } i=j \ v & ext{if } i
eq j \end{array} 
ight.$$

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

 $\overline{7}$ 

```
1
              if (in->log == inout->log)
2
                   c.val = in->val + inout->val;
3
              else
4
                   c.val = inout->val;
5
              c.log = inout->log;
6
              *inout = c;
7
              in++; inout++;
8
         }
9
     }
10
         Note that the inout argument to the user-defined function corresponds to the right-
11
     hand operand of the operator. When using this operator, we must be careful to specify that
12
     it is non-commutative, as in the following.
13
14
         int i,base;
15
         SegScanPair
                        a, answer;
16
         MPI_Op
                        myOp;
17
         MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
18
         MPI_Aint
                        disp[2];
19
          int
                        blocklen[2] = \{ 1, 1\};
20
         MPI_Datatype sspair;
21
22
         /* explain to MPI how type SegScanPair is defined
23
           */
24
         MPI_Get_address( &a, disp);
25
         MPI_Get_address( &a.log, disp+1);
26
         base = disp[0];
27
         for (i=0; i<2; ++i) disp[i] -= base;</pre>
28
         MPI_Type_create_struct( 2, blocklen, disp, type, &sspair );
29
         MPI_Type_commit( &sspair );
30
          /* create the segmented-scan user-op
31
           */
32
         MPI_Op_create(segScan, 0, &myOp);
33
          . . .
34
         MPI_Scan( &a, &answer, 1, sspair, myOp, comm );
35
36
37
             Nonblocking Collective Operations
     5.12
```

39 As described in Section 3.7, performance of many applications can be improved by over-40lapping communication and computation, and many systems enable this. Nonblocking 41 collective operations combine the potential benefits of nonblocking point-to-point opera-42tions, to exploit overlap and to avoid synchronization, with the optimized implementation 43 and message scheduling provided by collective operations [30, 34]. One way of doing this 44would be to perform a blocking collective operation in a separate thread. An alternative 45mechanism that often leads to better performance (e.g., avoids context switching, scheduler 46overheads, and thread management) is to use nonblocking collective communication [32].

<sup>47</sup> The nonblocking collective communication model is similar to the model used for non <sup>48</sup> blocking point-to-point communication. A nonblocking call initiates a collective operation,

### Unofficial Draft for Comment Only

which must be completed in a separate completion call. Once initiated, the operation may progress independently of any computation or other communication at participating processes. In this manner, nonblocking collective operations can mitigate possible synchronizing effects of collective operations by running them in the "background." In addition to enabling communication-computation overlap, nonblocking collective operations can perform collective operations on overlapping communicators, which would lead to deadlocks with blocking operations. Their semantic advantages can also be useful in combination with point-to-point communication.

As in the nonblocking point-to-point case, all calls are local and return immediately. irrespective of the status of other processes. The call initiates the operation, which indicates 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, 20i.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 communi-44cator. If the nonblocking call causes some system resource to be exhausted, then it will 4546fail and generate an MPI exception. Quality implementations of MPI should ensure that 47this happens only in pathological cases. That is, an MPI implementation should be able to 48 support a large number of pending nonblocking operations.

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9 10

11

12

13

1415

16

17

18

19

21

22

23 $^{24}$ 

25

26

27

28

2930

 $^{31}$ 

32

33

34

35

36

37

38 39

40

41

42

1 Unlike point-to-point operations, nonblocking collective operations do not match with  $\mathbf{2}$ blocking collective operations, and collective operations do not have a tag argument. All 3 processes must call collective operations (blocking and nonblocking) in the same order 4 per communicator. In particular, once a process calls a collective operation, all other 5processes in the communicator must eventually call the same collective operation, and no 6 other collective operation with the same communicator in between. This is consistent with  $\overline{7}$ the ordering rules for blocking collective operations in threaded environments.

8 9

10

11

12

13

Matching blocking and nonblocking collective operations is not allowed Rationale. because the implementation might use different communication algorithms for the two cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

14The use of tags for collective operations can prevent certain hardware optimizations. 15(End of rationale.) 16

17 Advice to users. If program semantics require matching blocking and nonblocking 18 collective operations, then a nonblocking collective operation can be initiated and 19immediately completed with a blocking wait to emulate blocking behavior. (End of 20advice to users.) 21

22In terms of data movements, each nonblocking collective operation has the same effect 23as its blocking counterpart for intracommunicators and intercommunicators after comple- $^{24}$ tion. Likewise, upon completion, nonblocking collective reduction operations have the same 25effect as their blocking counterparts, and the same restrictions and recommendations on 26reduction orders apply.

27The use of the "in place" option is allowed exactly as described for the corresponding 28blocking collective operations. When using the "in place" option, message buffers function 29as both send and receive buffers. Such buffers should not be modified or accessed until the 30 operation completes.

 $^{31}$ Progression rules for nonblocking collective operations are similar to progression of 32 nonblocking point-to-point operations, refer to Section 3.7.4.

Advice to implementors. Nonblocking collective operations can be implemented with local execution schedules [33] using nonblocking point-to-point communication and a reserved tag-space. (End of advice to implementors.)

- 5.12.1 Nonblocking Barrier Synchronization
- 39 40

33 34

35

36 37 38

 $^{41}$ MPI\_IBARRIER(comm, request) 42

42			
43	IN	comm	communicator (handle)
44	OUT	request	communication request (handle)
45			
46	int MPI	Ibarrier(MPI C	omm comm, MPI_Request *request)
47	-		, _ 1 1 1 1
48	MPI_Ibar	rier(comm, req	uest, ierror) BIND(C)

TYPE(MPI_Comm), INTENT(IN) :: comm	1
TYPE(MPI_Request), INTENT(OUT) :: request	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
	4
MPI_IBARRIER(COMM, REQUEST, IERROR)	5
INTEGER COMM, REQUEST, IERROR	6
MPI_IBARRIER is a nonblocking version of MPI_BARRIER. By calling MPI_IBARRIER,	7
a process notifies that it has reached the barrier. The call returns immediately, indepen-	8
dent of whether other processes have called MPI IBARRIER. The usual harrier semantics	9

a process notifies that it has reached the barrier. The call returns immediately, independent of whether other processes have called MPI\_IBARRIER. The usual barrier semantics are enforced at the corresponding completion operation (test or wait), which in the 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)
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)
OUT	request	communication request (handle)

MPI\_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C)
TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI\_Datatype), INTENT(IN) :: datatype
TYPE(MPI\_Comm), INTENT(IN) :: comm
TYPE(MPI\_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI\_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)

<type> BUFFER(\*)

INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR

This call starts a nonblocking variant of MPI\_BCAST (see Section 5.4).

```
1
      Example using MPI_IBCAST
\mathbf{2}
      The example in this section uses an intracommunicator.
3
4
     Example 5.24
5
          Start a broadcast of 100 ints from process 0 to every process in the group, perform some
6
      computation on independent data, and then complete the outstanding broadcast operation.
7
8
          MPI_Comm comm;
9
          int array1[100], array2[100];
10
          int root=0;
11
          MPI_Request req;
12
          . . .
13
          MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
14
          compute(array2, 100);
15
          MPI_Wait(&req, MPI_STATUS_IGNORE);
16
17
     5.12.3 Nonblocking Gather
18
19
20
      MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
21
                     request)
22
       IN
                  sendbuf
                                               starting address of send buffer (choice)
23
^{24}
       IN
                  sendcount
                                               number of elements in send buffer (non-negative inte-
25
                                               ger)
26
       IN
                  sendtype
                                               data type of send buffer elements (handle)
27
       OUT
                  recvbuf
                                               address of receive buffer (choice, significant only at
28
                                               root)
29
30
       IN
                                               number of elements for any single receive (non-negative
                  recvcount
^{31}
                                               integer, significant only at root)
32
       IN
                                               data type of recv buffer elements (significant only at
                  recvtype
33
                                               root) (handle)
34
                                               rank of receiving process (integer)
       IN
                  root
35
36
       IN
                                               communicator (handle)
                  comm
37
       OUT
                  request
                                               communication request (handle)
38
39
      int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
40
                     void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
41
                     MPI_Comm comm, MPI_Request *request)
42
43
     MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
44
                     root, comm, request, ierror) BIND(C)
45
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
48
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

	TYPE(MP	I_Comm), INTENT(IN) ::	comm	1	
	TYPE(MP	I_Request), INTENT(OUT)	:: request	2	
	INTEGER	, OPTIONAL, INTENT(OUT)	:: ierror	3	
MPT	TGATHER	(SENDBUF, SENDCOUNT, SE	NDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	4	
		ROOT, COMM, REQUEST,		5	
	<type></type>	<pre>SENDBUF(*), RECVBUF(*)</pre>		6 7	
			ECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	8	
	IERROR			9	
	This call	starts a nonblocking varian	t of MPI_GATHER (see Section $5.5$ ).	10	
	1110 0011	starts a nonsioning varian		11	
				12	
MPI	_IGATHE		dtype, recvbuf, recvcounts, displs, recvtype, root,	13	
		comm, request)		14	
IN	:	sendbuf	starting address of send buffer (choice)	15	
IN	:	sendcount	number of elements in send buffer (non-negative inte-	16	
			ger)	17 18	
IN	:	sendtype	data type of send buffer elements (handle)	19	
O	UT	recvbuf	address of receive buffer (choice, significant only at	20	
			root)	21	
IN		recvcounts	non-negative integer array (of length group size) con-	22	
111			taining the number of elements that are received from	23	
			each process (significant only at root)	24	
IN		displs	integer array (of length group size). Entry i specifies	25	
11 1		dishia	the displacement relative to recvbuf at which to place	26 27	
			the incoming data from process i (significant only at	27	
			root)	29	
IN		recvtype	data type of recv buffer elements (significant only at	30	
			root) (handle)	31	
IN		root	rank of receiving process (integer)	32	
				33	
IN		comm	communicator (handle)	34	
Ol	UT r	equest	communication request (handle)	35	
				36 37	
int	MPI_Iga		uf, int sendcount, MPI_Datatype sendtype,	38	
		-	<pre>int recvcounts[], const int displs[],</pre>	39	
			e, int root, MPI_Comm comm,	40	
		MPI_Request *request)		41	
MPI_	_Igather	v(sendbuf, sendcount, s	endtype, recvbuf, recvcounts, displs,	42	
			request, ierror) BIND(C)	43	
			(IN), ASYNCHRONOUS :: sendbuf	44	
		, DIMENSION(), ASYNCH		45	
		, INTENT(IN) :: sendco		46	
			<pre>OUS :: recvcounts(*), displs(*)</pre>	47 48	
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype48				

1 2		E(MPI_Comm), INTENT(I E(MPI_Request), INTEN	
3	INTE	EGER, OPTIONAL, INTEN	IT(OUT) :: ierror
4 5	MPI_IGAT	THERV(SENDBUF, SENDCO	UUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
6			, COMM, REQUEST, IERROR)
7	01	<pre>&gt; SENDBUF(*), RECVE</pre>	
8		GER SENDCOUNT, SENDI 1, REQUEST, IERROR	<pre>YPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,</pre>
9			
10 11	This	call starts a nonblockin	g variant of MPI_GATHERV (see Section $5.5$ ).
12	5.12.4	Nonblocking Scatter	
13	J.12.4	Nonbiocking Scatter	
14			
15 16	MPI_ISC/	ATTER(sendbuf, sendcou	nt, sendtype, recvbuf, recvcount, recvtype, root, comm,
17		request)	
18	IN	sendbuf	address of send buffer (choice, significant only at root)
19	IN	sendcount	number of elements sent to each process (non-negative
20 21			integer, significant only at root)
21 22 23	IN	sendtype	data type of send buffer elements (significant only at root) (handle)
24	OUT	recvbuf	address of receive buffer (choice)
25 26	IN	recvcount	number of elements in receive buffer (non-negative in-teger)
27 28	IN	recvtype	data type of receive buffer elements (handle)
29	IN	root	rank of sending process (integer)
30	IN	comm	communicator (handle)
31 32	OUT	request	communication request (handle)
33 34	int MPI	_Iscatter(const void*	sendbuf, int sendcount, MPI_Datatype sendtype,
35	_		int recvcount, MPI_Datatype recvtype, int root,
36		MPI_Comm comm,	MPI_Request *request)
37	MPI_Isca	atter(sendbuf, sendco	ount, sendtype, recvbuf, recvcount, recvtype,
38			quest, ierror) BIND(C)
39 40			INTENT(IN), ASYNCHRONOUS :: sendbuf
41			ASYNCHRONOUS :: recvbuf sendcount, recvcount, root
42			ENT(IN) :: sendtype, recvtype
43		E(MPI_Comm), INTENT(1	VI VI
44	TYPE	E(MPI_Request), INTEN	IT(OUT) :: request
45 46	INTE	EGER, OPTIONAL, INTEN	IT(OUT) :: ierror
47	MPI_ISCA	ATTER(SENDBUF, SENDCO	UUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
48		ROOT, COMM, REG	QUEST, IERROR)

			5
IN	sendbuf	address of send buffer (choice, significant only at root)	10
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	11 12
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	12 13 14 15
IN	sendtype	data type of send buffer elements (handle)	16
οι	•••	address of receive buffer (choice)	17 18
IN	recvcount	number of elements in receive buffer (non-negative in- teger)	19 20
IN	recvtype	data type of receive buffer elements (handle)	21
IN	root	rank of sending process (integer)	22 23
IN	comm	communicator (handle)	23 24
OL		communication request (handle)	25
00	JT request	communication request (nancie)	26
	<pre>int displs[], MPI_I int recvcount, MPI_ MPI_Request *reques Iscatterv(sendbuf, sendcount recvtype, root, con TYPE(*), DIMENSION(), INTE TYPE(*), DIMENSION(), ASYN</pre>	<pre>s, displs, sendtype, recvbuf, recvcount, m, request, ierror) BIND(C) NT(IN), ASYNCHRONOUS :: sendbuf CHRONOUS :: recvbuf ONOUS :: sendcounts(*), displs(*) count, root N) :: sendtype, recvtype : comm T) :: request</pre>	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
	RECVTYPE, ROOT, CON <type> SENDBUF(*), RECVBUF(* INTEGER SENDCOUNTS(*), DISPL COMM, REQUEST, IERROR</type>	S(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	42 43 44 45 46 47 48
	This can starts a nonbiocking var	iant of MPI_SCATTERV (see Section $5.6$ ).	10

 $\mathbf{2}$ 

	204		CHAPTER 5.	COLLECTIVE COMMUNICATION
1 2 3	5.12.5	Nonblocking Gather-to-a	II	
4 5	MPI_IA	LLGATHER(sendbuf, sendc request)	ount, sendtype, recvl	ouf, recvcount, recvtype, comm,
6 7	IN	sendbuf	starting addr	ess of send buffer (choice)
8 9	IN	sendcount	number of ele ger)	ements in send buffer (non-negative inte-
10	IN	sendtype	data type of	send buffer elements (handle)
11 12	OUT	recvbuf	address of re-	ceive buffer (choice)
13 14	IN	recvcount	number of el negative inte	ements received from any process (non- ger)
15	IN	recvtype	data type of	receive buffer elements (handle)
16 17	IN	comm	communicate	or (handle)
18	OUT	request	communicati	on request (handle)
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	<pre>int MPI_Iallgather(const void* sendbuf, int sendcount,</pre>			
<ol> <li>44</li> <li>45</li> <li>46</li> <li>47</li> <li>48</li> </ol>				

	t, sendtype, recvbuf, recvcounts, displs, recvtype, comm,	1 $2$	
sendbuf	starting address of send buffer (choice)	3	
sendcount	number of elements in send buffer (non-negative inte- ger)	4 5 6	
sendtype	data type of send buffer elements (handle)	7	
recvbuf	address of receive buffer (choice)	8	
recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process	9 10 11 12	
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	13 14 15	
recvtype	data type of receive buffer elements (handle)	16 17	
comm	communicator (handle)	18	
request	communication request (handle)	19	
<pre>OUT request communication request (handle) int MPI_Iallgatherv(const void* sendbuf, int sendcount,</pre>			
	<pre>request) sendbuf sendcount sendtype recvbuf recvcounts displs  recvtype comm request allgatherv(const void*     MPI_Datatype sendty     const int displs[],     MPI_Request* request atherv(sendbuf, sendcou     recvtype, comm, rec *), DIMENSION(), INTE *), DIMENSION(), ASYN ER, INTENT(IN) :: send ER, INTENT(IN) :: send ER, INTENT(IN), ASYNCHR MPI_Datatype), INTENT(I MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OU ER, OPTIONAL, INTENT(OU ER, OPTIONAL, INTENT(OU ER, SENDCOUNT, SENDTYPE, ST, IERROR</pre>	<pre>sendbuf starting address of send buffer (choice) sendcount number of elements in send buffer (non-negative inte- ger) sendtype data type of send buffer elements (handle) recvbuf address of receive buffer (choice) recvcounts non-negative integer array (of length group size) con- taining the number of elements that are received from each process 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 recvtype data type of receive buffer elements (handle) comm communicator (handle) request communicator request (handle) allgatherv(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) atherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) BIND(C) *), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf ER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) MPI_Comm), INTENT(IN) :: sendtype, recvtype MPI_Comm), INTENT(IN) :: sendtype, recvtuf, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) &gt; SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, ST, IERROR </pre>	

1 2 3	5.12.6	Nonblocking All-to-All	l Scatter/Gather		
4 5	MPI_IAI	_LTOALL(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype, comm, request)		
6 7	IN	sendbuf	starting address of send buffer (choice)		
8 9	IN	sendcount	number of elements sent to each process (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	<pre>int MPI_Ialltoall(const void* sendbuf, int sendcount,</pre>				
<ul> <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> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> <li>47</li> <li>48</li> </ul>	<ty INT</ty 	COMM, REQUEST pe> SENDBUF(*), REC EGER SENDCOUNT, SEN			

MPI_I	ALLTOALLV(sendbuf, sendcou recvtype, comm, req	nts, sdispls, sendtype, recvbuf, recvcounts, rdispls, uest)	12
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
OUT	recvbuf	address of receive buffer (choice)	11
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
OUT	request	communication request (handle)	22 23
<pre>int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const</pre>			
<pre>MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,</pre>			
T	nis call starts a nonblocking v	variant of $MPI_ALLTOALLV$ (see Section 5.8).	45 46 47 48

MPLIALITOALIV(conduct condequate edicale conditions recyclust recycluste reliance

1 $2$	MPI_IALL1	TOALLW(sendbuf, sendcounts, recvtypes, comm, request	sdispls, sendtypes, recvbuf, recvcounts, rdispls,		
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 15	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)		
16	OUT	recvbuf	address of receive buffer (choice)		
17 18 19	IN	recvcounts	integer array (of length group size) specifying the num- ber of elements that can be received from each rank (array of non-negative integers)		
20 21 22 23 24	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)		
25 26 27	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)		
28	IN	comm	communicator (handle)		
29 30	OUT	request	communication request (handle)		
31 32 33 34 35	<pre>int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], const</pre>				
36	MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,				
37 38 39 40	recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf				
41	<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recycounts(*), rdispls(*)</pre>				
42	recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),				
43	recvtypes(*)				
44	TYPE(MPI_Comm), INTENT(IN) :: comm				
45 46		MPI_Request), INTENT(OUT)	-		
40 47	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror		
48					

MPI_IALLTO		, SDISPLS, SENDTYPES, RECVBUF,	1
		RECVTYPES, COMM, REQUEST, IERROR)	2 3
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),</type>			
	S(*), RECVTYPES(*), COMM		4 5
			6
This ca	Il starts a nonblocking variar	t of MPI_ALLTOALLW (see Section $5.8$ ).	7
			8
5.12.7 No	nblocking Reduce		9
			10
			11
MPI_IREDU	ICE(sendbut, recvbut, count, d	atatype, op, root, comm, request)	12
IN	sendbuf	address of send buffer (choice)	13 14
OUT	recvbuf	address of receive buffer (choice, significant only at	14
		root)	16
IN	count	number of elements in send buffer (non-negative inte-	17
		ger)	18
IN	datatype	data type of elements of send buffer (handle)	19
IN	ор	reduce operation (handle)	20 21
IN	root	rank of root process (integer)	21
IN	comm	communicator (handle)	23
OUT	request	communication request (handle)	24
001	request	communication request (nancie)	25
int MPT Tr	educe(const void* sendhu	f, void* recvbuf, int count,	26
1110 111 1_11		e, MPI_Op op, int root, MPI_Comm comm,	27
	MPI_Request *request)		28 29
NDT T I			30
MP1_1reduc	ierror) BIND(C)	t, datatype, op, root, comm, request,	31
TVPF(*	), DIMENSION(), INTENT	(IN), ASYNCHRONOUS :: sendbuf	32
	<pre>&gt;, DIMENSION(), ASYNCH</pre>		33
	TR, INTENT(IN) :: count,		34
	<pre>IPI_Datatype), INTENT(IN)</pre>		35
	<pre>IPI_Op), INTENT(IN) :: o</pre>		36
	<pre>IPI_Comm), INTENT(IN) ::</pre>		37
	<pre>IPI_Request), INTENT(OUT)</pre>	-	38 39
INTEGE	ER, OPTIONAL, INTENT(OUT)	:: ierror	40
MPI_IREDUC	CE(SENDBUF, RECVBUF, COUN	T, DATATYPE, OP, ROOT, COMM, REQUEST,	41
	IERROR)		42
• 1	<pre>SENDBUF(*), RECVBUF(*)</pre>		43
INTEGE	ER COUNT, DATATYPE, OP, R	OOT, COMM, REQUEST, IERROR	44
This ca	ll starts a nonblocking variar	nt of MPI_REDUCE (see Section $5.9.1$ ).	45 46
Advice	e to implementors The im	plementation is explicitly allowed to use different	47
	-	cking reduction operations that might change the	48

```
210
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
           order of evaluation of the operations. However, as for MPI_REDUCE, it is strongly
\mathbf{2}
           recommended that MPI_IREDUCE be implemented so that the same result be obtained
3
           whenever the function is applied on the same arguments, appearing in the same order.
4
           Note that this may prevent optimizations that take advantage of the physical location
5
           of processes. (End of advice to implementors.)
6
           Advice to users. For operations which are not truly associative, the result delivered
7
           upon completion of the nonblocking reduction may not exactly equal the result deliv-
8
           ered by the blocking reduction, even when specifying the same arguments in the same
9
           order. (End of advice to users.)
10
11
12
     5.12.8 Nonblocking All-Reduce
13
14
15
     MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request)
16
       IN
                 sendbuf
                                             starting address of send buffer (choice)
17
18
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
19
       IN
                 count
                                             number of elements in send buffer (non-negative inte-
20
                                             ger)
21
       IN
                 datatype
                                             data type of elements of send buffer (handle)
22
23
       IN
                 ор
                                              operation (handle)
24
       IN
                 comm
                                             communicator (handle)
25
       OUT
                                              communication request (handle)
                 request
26
27
     int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,
28
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
29
                     MPI_Request *request)
30
^{31}
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
32
                     ierror) BIND(C)
33
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                       sendbuf
34
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
35
          INTEGER, INTENT(IN) :: count
36
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
          TYPE(MPI_Op), INTENT(IN) :: op
38
          TYPE(MPI_Comm), INTENT(IN) :: comm
39
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
40
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
41
     MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
42
                     IERROR)
43
          <type> SENDBUF(*), RECVBUF(*)
44
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
45
46
          This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6).
47
48
```

5.12. NONBLOCKING COLLECTIVE OPERATIONS 2			211
5.12.9	Nonblocking Reduce-Scatter	with Equal Blocks	1 2
MPI_I	REDUCE_SCATTER_BLOCK(sen	dbuf, recvbuf, recvcount, datatype, op, comm, req	3 uest) 4 5
IN	sendbuf	starting address of send buffer (choice)	6 7
OUT	- 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 dle)	
IN	ор	operation (handle)	13
IN	comm	communicator (handle)	14
OUT	request	communication request (handle)	15 16
<pre>int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>			
MPI_I < I	REQUEST, IERROR) type> SENDBUF(*), RECVBUF(*) NTEGER RECVCOUNT, DATATYPE, 'his call starts a nonblocking var	F, RECVBUF, RECVCOUNT, DATATYPE, OP, CON	33 34 35 36
			48

```
212
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.10
              Nonblocking Reduce-Scatter
\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
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
                 datatype
                                             data type of elements of input buffer (handle)
12
       IN
                                             operation (handle)
13
                 ор
14
       IN
                 comm
                                             communicator (handle)
15
       OUT
                 request
                                             communication request (handle)
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
```

# 5.12.11 Nonblocking Inclusive Scan

MPI_ISC	AN(sendbuf, recvbuf, cou	nt, datatype, op, comm, request)
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 MPI_		endbuf, void* recvbuf, int count, atatype, MPI_Op op, MPI_Comm comm, equest)
TYPH TYPH INTH TYPH TYPH TYPH INTH INTH <typ< td=""><td>BIND(C) E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Op), INTENT(IN) E(MPI_Comm), INTENT(I E(MPI_Request), INTEN EGER, OPTIONAL, INTEN AN(SENDBUF, RECVBUF, be&gt; SENDBUF(*), RECVE</td><td><pre>ENT(IN) :: datatype :: op EN) :: comm IT(OUT) :: request IT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre></td></typ<>	BIND(C) E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Op), INTENT(IN) E(MPI_Comm), INTENT(I E(MPI_Request), INTEN EGER, OPTIONAL, INTEN AN(SENDBUF, RECVBUF, be> SENDBUF(*), RECVE	<pre>ENT(IN) :: datatype :: op EN) :: comm IT(OUT) :: request IT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre>
This	call starts a nonblockin	g variant of $MPI_SCAN$ (see Section 5.11).

```
CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.12 Nonblocking Exclusive Scan
\mathbf{2}
3
4
     MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                             starting address of receive buffer (choice)
8
       IN
                 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)
12
                 op
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.
44
45
46
47
48
```

<pre>switch(rank) {</pre>	1
case 0:	2
<pre>MPI_Bcast(buf1, count, type, 0, comm);</pre>	3
MPI_Bcast(buf2, count, type, 1, comm);	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;
   case 2:
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 2:
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 2:
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 3:
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
   case 3:
   case
```

}

Assume that the group of comm0 is  $\{0,1\}$ , of comm1 is  $\{1, 2\}$  and of comm2 is  $\{2,0\}$ . If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependencies occur. Nonblocking collective operations can alleviate this issue.

### Example 5.27

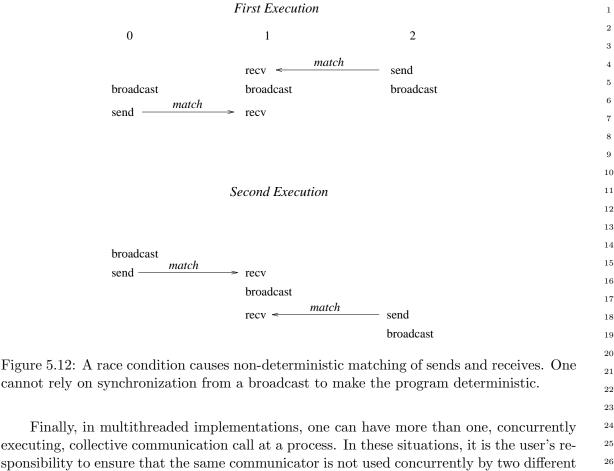
The following is erroneous.

 $^{24}$ 

 $41 \\ 42$ 

```
1
      switch(rank) {
\mathbf{2}
          case 0:
3
               MPI_Bcast(buf1, count, type, 0, comm);
4
               MPI_Send(buf2, count, type, 1, tag, comm);
5
               break:
6
          case 1:
7
               MPI_Recv(buf2, count, type, 0, tag, comm, status);
8
               MPI_Bcast(buf1, count, type, 0, comm);
9
               break;
10
     }
11
          Process zero executes a broadcast, followed by a blocking send operation. Process one
12
      first executes a blocking receive that matches the send, followed by broadcast call that
13
      matches the broadcast of process zero. This program may deadlock. The broadcast call on
14
      process zero may block until process one executes the matching broadcast call, so that the
15
      send is not executed. Process one will definitely block on the receive and so, in this case,
16
      never executes the broadcast.
17
          The relative order of execution of collective operations and point-to-point operations
18
     should be such, so that even if the collective operations and the point-to-point operations
19
      are synchronizing, no deadlock will occur.
20
21
      Example 5.28
22
          An unsafe, non-deterministic program.
23
^{24}
      switch(rank) {
25
          case 0:
26
               MPI_Bcast(buf1, count, type, 0, comm);
27
               MPI_Send(buf2, count, type, 1, tag, comm);
28
               break;
29
          case 1:
30
               MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
31
               MPI_Bcast(buf1, count, type, 0, comm);
32
               MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
33
               break;
34
          case 2:
35
               MPI_Send(buf2, count, type, 1, tag, comm);
36
               MPI_Bcast(buf1, count, type, 0, comm);
37
               break;
38
      }
39
40
          All three processes participate in a broadcast. Process 0 sends a message to process
41
      1 after the broadcast, and process 2 sends a message to process 1 before the broadcast.
42
      Process 1 receives before and after the broadcast, with a wildcard source argument.
43
          Two possible executions of this program, with different matchings of sends and receives,
44
      are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that
45
      a send executed after the broadcast is received at another node before the broadcast. This
```

<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.



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.

### **Unofficial Draft for Comment Only**

```
1
     MPI_Request req;
\mathbf{2}
3
     MPI_Ibarrier(comm, &req);
4
     MPI_Bcast(buf1, count, type, 0, comm);
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);
              MPI_Bcast(buf1, count, type, 0, dupcomm);
43
44
              MPI_Wait(&req, MPI_STATUS_IGNORE);
45
              break;
46
          case 1:
47
              MPI_Bcast(buf1, count, type, 0, dupcomm);
48
              MPI_Ibarrier(comm, &req);
```

```
MPI_Wait(&req, MPI_STATUS_IGNORE);
break;
```

}

Advice to users. The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

### Example 5.31

Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;
```

```
switch(rank) {
   case 0:
    MPI_Ibarrier(comm, &req);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    MPI_Send(buf, count, dtype, 1, tag, comm);
    break;
   case 1:
    MPI_Ibarrier(comm, &req);
    MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
```

```
}
```

The MPI library must progress the barrier in the MPI\_Recv call. Thus, the MPI\_Wait call in rank 0 will eventually complete, which enables the matching MPI\_Send so all calls eventually return.

## Example 5.32

Blocking and nonblocking collective operations do not match. The following example is erroneous.

```
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];
5
6
     switch(rank) {
7
8
          case 0:
9
            MPI_Ibarrier(comm, &reqs[0]);
            MPI_Send(buf, count, dtype, 1, tag, comm);
10
            MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
11
            break;
12
          case 1:
13
            MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
14
            MPI_Ibarrier(comm, &reqs[1]);
15
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, &reqs[1]);
^{31}
     compute(buf3);
32
     MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
33
     MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
34
35
           Advice to users. Pipelining and double-buffering techniques can efficiently be used
36
           to overlap computation and communication. However, having too many outstanding
37
           requests might have a negative impact on performance. (End of advice to users.)
38
39
                                      The use of pipelining may generate many outstanding
           Advice to implementors.
40
           requests. A high-quality hardware-supported implementation with limited resources
41
           should be able to fall back to a software implementation if its resources are exhausted.
42
           In this way, the implementation could limit the number of outstanding requests only
43
           by the available memory. (End of advice to implementors.)
44
45
46
     Example 5.35
47
48
```

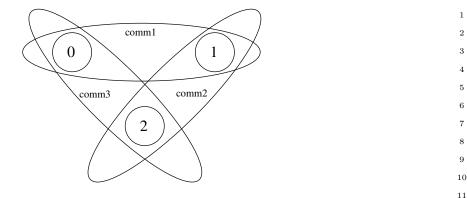


Figure 5.13: Example with overlapping communicators.

Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 5.13). The following example is started with three processes and three communicators. The first communicator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans ranks 0 and 2. It is not possible to perform a blocking collective operation on all communicators because there exists no deadlock-free order to invoke them. However, nonblocking collective operations can easily be used to achieve this task.

```
MPI_Request reqs[2];
```

```
switch(rank) {
    case 0:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
                                                                                  27
      break;
                                                                                  28
    case 1:
                                                                                  29
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
                                                                                  30
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
      break;
                                                                                  33
    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]);
                                                                                  35
      break;
                                                                                  36
}
                                                                                  37
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
                                                                                  38
```

Advice to users. This method can be useful if overlapping neighboring regions (halo or ghost zones) are used in collective operations. The sequence of the two calls in each process is irrelevant because the two nonblocking operations are performed on different communicators. (End of advice to users.)

### Example 5.36

The progress of multiple outstanding nonblocking collective operations is completely independent.

### **Unofficial Draft for Comment Only**

1213 14

15

16

17

18

19

2021

22 23

 $^{24}$ 25

26

31

32

39

40

41

42

43 4445

46

47

```
1
     MPI_Request reqs[2];
\mathbf{2}
3
      compute(buf1);
4
     MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
\mathbf{5}
      compute(buf2);
6
     MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
7
     MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
8
      /* nothing is known about the status of the first bcast here */
9
     MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
10
11
          Finishing the second MPI_IBCAST is completely independent of the first one. This
      means that it is not guaranteed that the first broadcast operation is finished or even started
12
      after the second one is completed via reqs[1].
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

## Chapter 6

# Groups, Contexts, Communicators, and Caching

## 6.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [56] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

#### 6.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments. 

 $^{24}$ 

## 6.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- Groups of processes,
- Virtual topologies,
- Attribute caching,
- Communicators.

<sup>13</sup> <sup>14</sup> **Communicators** (see [21, 54, 58]) 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 a par with MPI built-in features. This <sup>21</sup> can 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

45 46

Intra-communicators. The most commonly used means for message passing in MPI is via
 intra-communicators. Intra-communicators contain an instance of a group, contexts of
 communication for both point-to-point and collective communication, and the ability to
 include virtual topology and other attributes. These features work as follows:

• **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.

- **Groups** define the participants in the communication (see above) of a communicator.
- 47 48

- A virtual topology defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 7 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. This practice can be followed in MPI by using the predefined communicator MPI\_COMM\_WORLD. Users who are satisfied with this practice can plug in MPI\_COMM\_WORLD wherever a communicator argument is required, and can consequently disregard the rest of this chapter. (End of advice to users.)

Inter-communicators. The discussion has dealt so far with intra-communication: communication within a group. MPI also supports inter-communication: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- Contexts provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in an related manner to intracommunicators. Users who do not need inter-communication in their applications can safely 

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

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

## 6.2 Basic Concepts

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

6.2.1 Groups

1

 $\mathbf{2}$ 

3 4

5 6

> 7 8

9

20 21

22

23 24

25

26

27

28

34

35

<sup>10</sup> A **group** is an ordered set of process identifiers (henceforth processes); processes are <sup>11</sup> implementation-dependent objects. Each process in a group is associated with an inte-<sup>12</sup> ger **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque <sup>13</sup> **group objects**, and hence cannot be directly transferred from one process to another. A <sup>14</sup> group is used within a communicator to describe the participants in a communication "uni-<sup>15</sup> verse" and to rank such participants (thus giving them unique names within that "universe" <sup>16</sup> of communication).

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

Advice to users. MPI\_GROUP\_EMPTY, which is a valid handle to an empty group, should not be confused with MPI\_GROUP\_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (*End of advice to users.*)

Advice to implementors. A group may be represented by a virtual-to-real processaddress-translation table. Each communicator object (see below) would have a pointer to such a table.

Simple implementations of MPI will enumerate groups, such as in a table. However,
 more advanced data structures make sense in order to improve scalability and memory
 usage with large numbers of processes. Such implementations are possible with MPI.
 *(End of advice to implementors.)*

## 6.2.2 Contexts

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

 $41 \\ 42$ 

43

44

45

46

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicatorgenerating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

#### 6.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 7), communicators may also "cache" additional information (see Section 6.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

#### 6.2.4 Predefined Intra-Communicators

An initial intra-communicator MPI\_COMM\_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI\_INIT or MPI\_INIT\_THREAD has been called. In addition, the communicator MPI\_COMM\_SELF is provided, which includes only the process itself.

The predefined constant MPI\_COMM\_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the 37 computation are available after MPI is initialized. For this case, MPI\_COMM\_WORLD is a 3839 communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynami-40 41 cally join an MPI execution, it may be the case that a process starts an MPI computation 42without having access to all other processes. In such situations, MPI\_COMM\_WORLD is a communicator incorporating all processes with which the joining process can immediately 4344communicate. Therefore, MPI\_COMM\_WORLD may simultaneously represent disjoint groups 45in different processes.

All MPI implementations are required to provide the MPI\_COMM\_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to this communicator does not appear as a pre-defined constant, but it may be accessed using

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28 29

30

31

32 33

34

35

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

5 6 7

8

9

10 11

## 6.3 Group Management

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

```
6.3.1 Group Accessors
12
13
14
     MPI_GROUP_SIZE(group, size)
15
16
       IN
                                            group (handle)
                 group
17
       OUT
                size
                                            number of processes in the group (integer)
18
19
     int MPI_Group_size(MPI_Group group, int *size)
20
21
     MPI_Group_size(group, size, ierror) BIND(C)
22
          TYPE(MPI_Group), INTENT(IN) :: group
23
          INTEGER, INTENT(OUT) :: size
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
25
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
26
          INTEGER GROUP, SIZE, IERROR
27
28
29
     MPI_GROUP_RANK(group, rank)
30
^{31}
       IN
                                            group (handle)
                 group
32
       OUT
                 rank
                                            rank of the calling process in group,
                                                                                        or
33
                                            MPI_UNDEFINED if the process is not a member (in-
34
                                            teger)
35
36
     int MPI_Group_rank(MPI_Group group, int *rank)
37
38
     MPI_Group_rank(group, rank, ierror) BIND(C)
39
          TYPE(MPI_Group), INTENT(IN) :: group
40
          INTEGER, INTENT(OUT) :: rank
41
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
42
     MPI_GROUP_RANK(GROUP, RANK, IERROR)
43
          INTEGER GROUP, RANK, IERROR
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 ${\sf ranks1}$ and ${\sf ranks2}$ 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	
		MPI_UNDEFINED when no correspondence exists.	9	
			10	
int MPI_(	Froup_translate_ranks(MPI MPI_Group group2, in	_Group group1, int n, const int ranks1[],	11 12	
NDT 0			13	
MP1_Group	o_translate_ranks(group1, BIND(C)	n, ranks1, group2, ranks2, ierror)	14	
TYPE	(MPI_Group), INTENT(IN) :	: group1, group2	15	
	GER, INTENT(IN) :: n, ra		16 17	
	GER, INTENT(OUT) :: rank		18	
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	19	
MPI_GROUN	P_TRANSLATE_RANKS(GROUP1,	N, RANKS1, GROUP2, RANKS2, IERROR)	20	
INTEC	GER GROUP1, N, RANKS1(*),	GROUP2, RANKS2(*), IERROR	21	
This f	function is important for determined	mining the relative numbering of the same processes	22 23	
		ne knows the ranks of certain processes in the group	20	
of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group.			25	
	PROC_NULL is a valid rank for i PI_PROC_NULL as the translat	nput to MPI_GROUP_TRANSLATE_RANKS, which	26	
	-I_FROC_NOLL as the translat	eu raink.	27 28	
	UP_COMPARE(group1, group2	2, result)	29 30	
IN	group1	first group (handle)	31	
IN	group2	second group (handle)	32	
OUT	result	result (integer)	33	
			34 35	
int MPI_(	Group_compare(MPI_Group g	roup1,MPI_Group group2, int *result)	36	
MPI_Group	_compare(group1, group2,	result, ierror) BIND(C)	37	
-	(MPI_Group), INTENT(IN) :		38	
	SER, INTENT(OUT) :: resu		39	
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	40	
MPI_GROUN	_COMPARE(GROUP1, GROUP2,	RESULT, IERROR)	41 42	
INTEC	GER GROUP1, GROUP2, RESUL	T, IERROR	43	
MPI_IDEN	MPI_IDENT results if the group members and group order is exactly the same in both groups.			
This happ	This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if <sup>4</sup>			
the group	members are the same but the	order is different. $MPI\_UNEQUAL$ results otherwise.	46	
	47			

## Unofficial Draft for Comment Only

```
6.3.2 Group Constructors
```

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.)

```
21
22
23
```

 $^{24}$ 

1

 $\mathbf{2}$ 

11

12

13

14

15

16 17

18

19

20

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 34 MPI\_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

<pre>MPI_Group_union(group1, group2, newgroup, ierror) BIND(C)     TYPE(MPI_Group), INTENT(IN) :: group1, group2     TYPE(MPI_Group), INTENT(OUT) :: newgroup</pre>				
INTEG	GER, OPTIONAL, INTENT(OUT	) :: ierror	4 5	
	MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR			
	,,		7 8	
			9	
MPI_GRO	UP_INTERSECTION(group1, g	group2, newgroup)	10	
IN	group1	first group (handle)	11	
IN	group2	second group (handle)	12	
OUT	newgroup	intersection group (handle)	13	
001	newgroup	intersection group (nandle)	14 15	
int MPT (	Froun intersection(MPI Gr	oup group1, MPI_Group group2,	16	
1110 111 1_1	MPI_Group *newgroup)		17	
MDT Garage			18	
-	(MPI_Group), INTENT(IN) :	oup2, newgroup, ierror) BIND(C)	19	
	(MPI_Group), INTENT(OUT)		20	
	GER, OPTIONAL, INTENT(OUT	<b>.</b>	21	
			22 23	
	P_INTERSECTION(GROUP1, GR GER GROUP1, GROUP2, NEWGR		24	
1010			25	
			26	
MPI_GRO	UP_DIFFERENCE(group1, gro	up2, newgroup)	27	
IN	group1	first group (handle)	28	
IN	group2	second group (handle)	29 30	
	<b>C</b> .		31	
OUT	newgroup	difference group (handle)	32	
int MDT (	Smour difference (MDI Crou	n maying MDI (mayin maying)	33	
int MP1_0	MPI_Group *newgroup)	p group1, MPI_Group group2,	34	
			35	
		p2, newgroup, ierror) BIND(C)	36	
	<pre>(MPI_Group), INTENT(IN) : (MPI_Group), INTENT(OUT)</pre>		37 38	
	GER, OPTIONAL, INTENT(OUT)		зо 39	
			40	
	P_DIFFERENCE(GROUP1, GROU		41	
INTEC	GER GROUP1, GROUP2, NEWGR	UUP, IERRUR	42	
The set-lik	te operations are defined as fo	llows:	43	
	union All elements of the first group (group1), followed by all elements of second group (group2) not in first.			
	ntersect all elements of the first group that are also in the second group, ordered as in first group.			

1difference 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

int MPI_(	<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>			
-	<pre>MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C)     TYPE(MPI_Group), INTENT(IN) :: group</pre>			
	GER, INTENT(IN) :: n, rai	<b>.</b>	5	
	(MPI_Group), INTENT(OUT)		6	
	GER, OPTIONAL, INTENT(OUT)	<b>o i</b>	7 8	
	P_EXCL(GROUP, N, RANKS, N		9	
	GER GROUP, N, RANKS, N	-	10	
			11	
		eates a group of processes newgroup that is obtained	12	
-		with ranks ranks[0] , ranks[n-1]. The ordering of ordering in group. Each of the n elements of ranks	13	
-		ements must be distinct; otherwise, the program is	14 15	
	If $n = 0$ , then newgroup is i		15	
			17	
	UP_RANGE_INCL(group, n, ra	ngos nowgroup)	18	
		,	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 23	
		(first rank, last rank, stride) indicating ranks in $group$	23 24	
		of processes to be included in newgroup	25	
OUT	newgroup	new group derived from above, in the order defined by	26	
		ranges (handle)	27	
			28	
int MPI_(	<pre>int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],</pre>			
	MPI_Group *newgroup)		30 31	
-	<pre>MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)</pre>			
	TYPE(MPI_Group), INTENT(IN) :: group			
	GER, INTENT(IN) :: n, rai		34	
	TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
	ER, OFIIONAL, INIENI(001	) :: ierror	36	
	P_RANGE_INCL(GROUP, N, RA		37	
INTE	GER GROUP, N, RANGES(3,*)	, NEWGROUP, IERROR	38	
If ranges	consist of the triplets		39 40	
(fir	$st_1, last_1, stride_1), \dots, (first_n, last_n)$	last stride)	41	
(j v)	(j, i, i, j, j, i,	$use_n, ser ue_n)$	42	
then newg	then newgroup consists of the sequence of processes in group with ranks		43	
<i>c</i> ·		$  last_1 - first_1  $	44	
firs	$t_1, first_1 + stride_1,, first_1 -$	+ $\begin{bmatrix} \hline stride_1 \end{bmatrix}$ stride <sub>1</sub> ,	45	
		last first	46 47	
firs	$first_n, first_n + stride_n,, first_n + \left  \frac{last_n - first_n}{stride_n} \right  stride_n.$			
	Unofficial Dr	oft for Commont Only		

<sup>1</sup> Each computed rank must be a valid rank in group and all computed ranks must be <sup>2</sup> distinct, or else the program is erroneous. Note that we may have  $first_i > last_i$ , and  $stride_i$ <sup>3</sup> may be negative, but cannot be zero. <sup>4</sup> The functionality of this routine is specified to be equivalent to expanding the array

<sup>5</sup> of ranges to an array of the included ranks and passing the resulting array of ranks and <sup>6</sup> other arguments to MPI\_GROUP\_INCL. A call to MPI\_GROUP\_INCL is equivalent to a call <sup>7</sup> to MPI\_GROUP\_RANGE\_INCL with each rank i in ranks replaced by the triplet (i,i,1) in <sup>8</sup> the argument ranges.

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

9 10

11

31

32

40

45

12	IN	group	group (handle)
13 14	IN	n	number of elements in array ranges (integer)
15	IN	ranges	a one-dimensional array of integer triplets of the form
16			(first rank, last rank, stride), indicating the ranks in
17 18			group of processes to be excluded from the output group newgroup.
19	OUT	newgroup	new group derived from above, preserving the order
20			in group (handle)
21			
22	int MPI_G	roup_range_excl(MPI_Group	group, int n, int ranges[][3],
23 24		MPI_Group *newgroup)	
25	MPI_Group	_range_excl(group, n, ran	ges, newgroup, ierror) BIND(C)
26	-	MPI_Group), INTENT(IN) ::	<b>o o i</b>
27	INTEG	ER, INTENT(IN) :: n, ran	ges(3,n)
28		MPI_Group), INTENT(OUT) :	<b>.</b>
29	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror
30			

```
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,
 or else the program is erroneous.

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

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

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

- - -

~

\_

6.3.3 Group Destructors	
MPI_GROUP_FREE(group) INOUT group	group (handle)
<pre>int MPI_Group_free(MPI_Group *grou</pre>	•
<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.)

nication. (End of advice to implementors.)		35
		36
6.4.1 Communicator Accessors		37
		38
The following are all local operations.		39
		40
		41
MPI_COMM_SIZE(comm, size)		42
IN comm	communicator (handle)	43
OUT size	number of processes in the group of <b>comm</b> (integer)	44
	nameer of proceeder in the group of comm (integer)	45
int MPI_Comm_size(MPI_Comm comm, int *size)		47
MPI_Comm_size(comm, size, ierror) BIND(C)		48

```
Unofficial Draft for Comment Only
```

 $\frac{24}{25}$ 

```
1
          TYPE(MPI_Comm), INTENT(IN) ::
                                              comm
\mathbf{2}
          INTEGER, INTENT(OUT) :: size
3
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
4
     MPI_COMM_SIZE(COMM, SIZE, IERROR)
5
          INTEGER COMM, SIZE, IERROR
6
7
                       This function is equivalent to accessing the communicator's group with
           Rationale.
8
           MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and
9
           then freeing the temporary group via MPI_GROUP_FREE. However, this function is
10
           so commonly used, that this shortcut was introduced. (End of rationale.)
11
12
           Advice to users.
                               This function indicates the number of processes involved in a
13
           communicator. For MPI_COMM_WORLD, it indicates the total number of processes
14
           available (for this version of MPI, there is no standard way to change the number of
15
           processes once initialization has taken place).
16
           This call is often used with the next call to determine the amount of concurrency
17
           available for a specific library or program. The following call, MPI_COMM_RANK
18
           indicates the rank of the process that calls it in the range from 0 \dots size - 1, where size
19
           is the return value of MPI_COMM_SIZE.(End of advice to users.)
20
21
22
23
     MPI_COMM_RANK(comm, rank)
24
       IN
                                              communicator (handle)
                 comm
25
       OUT
                                              rank of the calling process in group of comm (integer)
                 rank
26
27
     int MPI_Comm_rank(MPI_Comm comm, int *rank)
28
29
     MPI_Comm_rank(comm, rank, ierror) BIND(C)
30
          TYPE(MPI_Comm), INTENT(IN) ::
                                              comm
^{31}
          INTEGER, INTENT(OUT) :: rank
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_COMM_RANK(COMM, RANK, IERROR)
          INTEGER COMM, RANK, IERROR
35
36
37
           Rationale. This function is equivalent to accessing the communicator's group with
38
           MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK,
39
           and then freeing the temporary group via MPI_GROUP_FREE. However, this function
40
           is so commonly used, that this shortcut was introduced. (End of rationale.)
41
           Advice to users. This function gives the rank of the process in the particular commu-
42
           nicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.
43
44
           Many programs will be written with the master-slave model, where one process (such
45
           as the rank-zero process) will play a supervisory role, and the other processes will
46
           serve as compute nodes. In this framework, the two preceding calls are useful for
47
           determining the roles of the various processes of a communicator. (End of advice to
48
           users.)
```

MPI\_COMM\_COMPARE(comm1, comm2, result)

IN	comm1	first communicator (handle)	
IN	comm2	second communicator (handle)	
OUT	result	result (integer)	
MPI_Comm_	Comm_compare(MPI_Comm comm compare(comm1, comm2, res MPI_Comm), INTENT(IN) ::	-	1
	ER, INTENT(OUT) :: resul ER, OPTIONAL, INTENT(OUT)		1
			1
	MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) INTEGER COMM1, COMM2, RESULT, IERROR		
	regulta if and only if comm1 as	nd comm? are handles for the same object (identical	

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: <sup>33</sup> 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. <sup>36</sup> <sup>37</sup> <sup>38</sup>

An intracommunicator involves a single group while an intercommunicator involves two groups. Where the following discussions address intercommunicator semantics, the two groups in an intercommunicator are called the *left* and *right* groups. A process in an intercommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the *local* group; the other group (relative to that process) is the *remote* group. The left and right group labels give us a way to describe the two groups in an intercommunicator that is not relative to any particular process (as the local and remote groups are). 

 $^{31}$ 

1 MPI\_COMM\_DUP(comm, newcomm) 2 IN comm communicator (handle) 3 OUT copy of comm (handle) newcomm 4 56 int MPI\_Comm\_dup(MPI\_Comm comm, MPI\_Comm \*newcomm) 7 MPI\_Comm\_dup(comm, newcomm, ierror) BIND(C) 8 TYPE(MPI\_Comm), INTENT(IN) :: comm 9 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 12MPI\_COMM\_DUP(COMM, NEWCOMM, IERROR) 13 INTEGER COMM, NEWCOMM, IERROR 14MPI\_COMM\_DUP Duplicates the existing communicator comm with associated key 15values, topology information, and info hints. For each key value, the respective copy callback 16function determines the attribute value associated with this key in the new communicator; 17one particular action that a copy callback may take is to delete the attribute from the new 18 communicator. Returns in newcomm a new communicator with the same group or groups, 19 same topology, same info hints, any copied cached information, but a new context (see 20Section 6.7.1). 2122 Advice to users. This operation is used to provide a parallel library call with a dupli-23cate communication space that has the same properties as the original communicator. 24This includes any attributes (see below), topologies (see Chapter 7), and associated 25info hints (see Section 6.4.4). This call is valid even if there are pending point-to-point 26communications involving the communicator comm. A typical call might involve a 27MPI\_COMM\_DUP at the beginning of the parallel call, and an MPI\_COMM\_FREE of 28that duplicated communicator at the end of the call. Other models of communicator 29 management are also possible. 30 This call applies to both intra- and inter-communicators. (End of advice to users.) 3132 Advice to implementors. One need not actually copy the group information, but only 33 add a new reference and increment the reference count. Copy on write can be used 34 for the cached information. (End of advice to implementors.) 35 36 37 38MPI\_COMM\_DUP\_WITH\_INFO(comm, info, newcomm) 39 IN comm communicator (handle) 40 IN info info object (handle) 41 42OUT newcomm copy of comm (handle) 43 44int MPI\_Comm\_dup\_with\_info(MPI\_Comm comm, MPI\_Info info, MPI\_Comm \*newcomm) 4546MPI\_Comm\_dup\_with\_info(comm, info, newcomm, ierror) BIND(C) 47TYPE(MPI\_Comm), INTENT(IN) :: comm 48 TYPE(MPI\_Info), INTENT(IN) :: info

	(MPI_Comm), INTENT(OUT) :: GER, OPTIONAL, INTENT(OUT)		1 2
MDT COMM	MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)		
	_DOP_WITH_INFO(COMM, INFO, GER COMM, INFO, NEWCOMM, I	-	4
INTEC	ALK COMM, INFO, NEWCOMM,	TERIOR	5
		haves exactly as $MPI_{COMM}_{DUP}$ except that the	6
		cator comm are not duplicated in newcomm. The	7
-	hints provided by the argument info are associated with the output communicator newcomm <sup>8</sup>		
instead.			9 10
Data	angle. It is am acted that som	a hinta will only be walted at communication exection	10
	-	he hints will only be valid at communicator creation most communicator creation calls do not provide	12
	, 8	ate info hints with a duplicate of any communicator	13
	0	MPI_COMM_DUP_WITH_INFO. ( <i>End of rationale.</i> )	14
	easien enne enne agn a can ee		15
			16
			17
	M_IDUP(comm, newcomm, red	quest)	18
IN	comm	communicator (handle)	19
OUT	newcomm	copy of comm (handle)	20
OUT	request	communication request (handle)	21 22
	•		22
int MPI_(	Comm_idup(MPI_Comm comm, N	MPI_Comm *newcomm, MPI_Request *request)	24
MPI_Comm_	_idup(comm, newcomm, reque	est, ierror) BIND(C)	25
TYPE	(MPI_Comm), INTENT(IN) ::	comm	26
TYPE	(MPI_Comm), INTENT(OUT) ::	: newcomm	27
TYPE	(MPI_Request), INTENT(OUT)	) :: request	28 29
INTEC	GER, OPTIONAL, INTENT(OUT)	) :: ierror	30
MPI_COMM	_IDUP(COMM, NEWCOMM, REQU	EST, IERROR)	31
	GER COMM, NEWCOMM, REQUEST	-	32
		a maint of MDL COMM DUD. The concention of	33
		g variant of MPI_COMM_DUP. The semantics of M_DUP was executed at the time that	34
		e, attributes changed after MPI_COMM_IDUP will	35
	-	r. All restrictions and assumptions for nonblock-	36
-		12) apply to MPI_COMM_IDUP and the returned	37
request.	the operations (see Section of		38
-	rroneous to use the communic	ator <b>newcomm</b> as an input argument to other MPI	39
	before the MPI_COMM_IDUP		40
			41 42
	-	crucial for the development of purely nonblocking	42 43
libra	ries (see $[36]$ ). (End of rational	ale.)	43
			45
			46
			47
			48

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

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

```
2
       IN
                                              communicator (handle)
                 comm
3
       IN
                                              Group, which is a subset of the group of
                 group
4
                                              comm (handle)
5
6
       OUT
                 newcomm
                                              new communicator (handle)
7
8
     int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
9
     MPI_Comm_create(comm, group, newcomm, ierror) BIND(C)
10
          TYPE(MPI_Comm), INTENT(IN) ::
                                              comm
11
          TYPE(MPI_Group), INTENT(IN) ::
                                               group
12
          TYPE(MPI_Comm), INTENT(OUT) ::
                                               newcomm
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
16
          INTEGER COMM, GROUP, NEWCOMM, IERROR
17
     If comm is an intracommunicator, this function returns a new communicator newcomm with
18
     communication group defined by the group argument. No cached information propagates
19
     from comm to newcomm. Each process must call with a group argument that is a subgroup
20
21
     of the group associated with comm; this could be MPI_GROUP_EMPTY. The processes may
     specify different values for the group argument. If a process calls with a non-empty group
22
     then all processes in that group must call the function with the same group as argument,
23
     that is the same processes in the same order. Otherwise the call is erroneous. This implies
^{24}
     that the set of groups specified across the processes must be disjoint. If the calling process
25
     is a member of the group given as group argument, then newcomm is a communicator with
26
     group as its associated group. In the case that a process calls with a group to which it does
27
     not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The
28
     function is collective and must be called by all processes in the group of comm.
29
30
           Rationale.
                       The interface supports the original mechanism from MPI-1.1, which re-
31
           quired 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.)
35
36
                       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
42
                groups are used to create new communicators.
43
             • It permits implementations sometimes to avoid communication related to context
44
                creation.
45
46
           (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) further to subdivide a computation into parallel sub-computations. A more general service is provided by MPI\_COMM\_SPLIT, below. (*End of advice to users.*)

Advice to implementors. When calling MPI\_COMM\_DUP, all processes call with the same group (the group associated with the communicator). When calling MPI\_COMM\_CREATE, the processes provide the same group or disjoint subgroups. For both calls, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of newcomm. All processes in the same local group of comm must specify the same value for group, i.e., the same members in the same order. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI\_COMM\_NULL is returned.

*Rationale.* In the case where either the left or right group is empty, a null communicator is returned instead of an intercommunicator with MPI\_GROUP\_EMPTY because the side with the empty group must return MPI\_COMM\_NULL. (*End of rationale.*)

**Example 6.1** The following example illustrates how the first node in the left side of an intercommunicator could be joined with all members on the right side of an intercommunicator to form a new intercommunicator.

```
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

 $\frac{24}{25}$ 

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
                                    IŌ
                                                                     2
9
10
11
                                  1
                                                                 ١
                                                               ١
                        After
12
                                   I
13
                                 1
14
                               00
15
                                                                   1
16
17
                                                                        6
                                                                  2
18
19
20
21
22
     Figure 6.1: Intercommunicator create using MPI_COMM_CREATE extended to intercom-
23
     municators. The input groups are those in the grey circle.
^{24}
25
                 MPI_Comm_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
37
     MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)
38
       IN
                 comm
                                             intracommunicator (handle)
39
       IN
                                             group, which is a subset of the group of comm (handle)
                 group
40
41
                                              "safe" tag (integer)
       IN
                 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
```

TYPE(MPI_Group), INTENT(IN) :: group	1
INTEGER, INTENT(IN) :: tag TYPE(MPI_Comm), INTENT(OUT) :: newcomm	2 3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	5
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)	6
INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR	7
MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however,	8
MPI_COMM_CREATE must be called by all processes in the group of	9 10
comm, whereas MPI_COMM_CREATE_GROUP must be called by all processes in group, which is a subgroup of the group of comm. In addition, MPI_COMM_CREATE_GROUP	11
requires that comm is an intracommunicator. MPI_COMM_CREATE_GROUP returns a new	12
intracommunicator, newcomm, for which the group argument defines the communication	13
group. No cached information propagates from comm to newcomm. Each process must	14
provide a group argument that is a subgroup of the group associated with comm; this	15
could be MPI_GROUP_EMPTY. If a non-empty group is specified, then all processes in that	16
group must call the function, and each of these processes must provide the same arguments,	17 18
including a group that contains the same members with the same ordering. Otherwise	18
the call is erroneous. If the calling process is a member of the group given as the group argument, then newcomm is a communicator with group as its associated group. If the	20
calling process is not a member of group, e.g., group is MPI_GROUP_EMPTY, then the call	21
is a local operation and MPI_COMM_NULL is returned as newcomm.	22
	23
Rationale. Functionality similar to MPI_COMM_CREATE_GROUP can be imple-	24
mented through repeated MPI_INTERCOMM_CREATE and	25 26
MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communica- tors at each process in group and build up an intracommunicator with group	20 27
group [16]. Such an algorithm requires the creation of many intermediate communi-	28
cators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation	29
that avoids this overhead. (End of rationale.)	30
	31
Advice 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 MPI_COMM_CREATE_GROUP and using that communicator as the local communi-	33 34
cator argument to MPI_INTERCOMM_CREATE. (End of advice to users.)	35
	36
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
Advice to users. MPI_COMM_CREATE may provide lower overhead than	41 42
MPI_COMM_CREATE_GROUP because it can take advantage of collective communi-	42 43
cation on comm when constructing newcomm. (End of advice to users.)	44
	45
	46
	47
	48

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

1

 $^{31}$ 

2	INI		· · · · · · · · · · · · · · · · · · ·
3	IN	comm	communicator (handle)
4	IN	color	control of subset assignment (integer)
5	IN	key	control of rank assignment (integer)
6	OUT	newcomm	new communicator (handle)
7			
8 9	int MPI_0	Comm_split(MPI_Comm comm,	<pre>int color, int key, MPI_Comm *newcomm)</pre>
10	MPI_Comm	_split(comm, color, key, n	newcomm, ierror) BIND(C)
11	TYPE	(MPI_Comm), INTENT(IN) ::	comm
12	INTE	GER, INTENT(IN) :: color	, key
13	TYPE	(MPI_Comm), INTENT(OUT) :	: newcomm
14	INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror
15	МРТ СОММ	_SPLIT(COMM, COLOR, KEY, 1	
16		GER COMM, COLOR, KEY, NEW	
17			
18			ciated with comm into disjoint subgroups, one for
19			tains all processes of the same color. Within each
20 21	• • •	-	the order defined by the value of the argument
21		_	r rank in the old group. A new communicator is
22			in newcomm. A process may supply the color value
23 24			mm returns MPI_COMM_NULL. This is a collective
24 25	,		ovide different values for color and key.
25 26			a call to MPI_COMM_CREATE(comm, group, new-
	,	-	COMM_SPLIT(comm, color, key, newcomm), where
27	processes	that are members of their grou	up argument provide $color = number of the group$

<sup>25</sup> With an intracommunicator comm, a call to MPI\_COMM\_CREATE(comm, group, new-<sup>26</sup> comm) is equivalent to a call to MPI\_COMM\_SPLIT(comm, color, key, newcomm), where <sup>27</sup> processes that are members of their group argument provide color = number of the group <sup>28</sup> (based on a unique numbering of all disjoint groups) and key = rank in group, and all <sup>29</sup> processes that are not members of their group argument provide color = MPI\_UNDEFINED. <sup>30</sup> The value of color must be non-negative.

This is an extremely powerful mechanism for dividing a single Advice to users. 32 communicating group of processes into k subgroups, with k chosen implicitly by the 33 user (by the number of colors asserted over all the processes). Each resulting com-34 municator will be non-overlapping. Such a division could be useful for defining a 35hierarchy 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 40case, 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 re-43 quired to discover group membership. MPI\_COMM\_CREATE\_GROUP is useful when 44 all processes in a given group have complete information of the members of their group 45 and synchronization 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 doesn't 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.

```
/* Client code */
MPI_Comm multiple_server_comm;
MPI_Comm single_server_comm;
int
          color, rank, num_servers;
                                                                         34
/* Create intercommunicator with clients and servers:
                                                                         35
                                                                         36
   multiple_server_comm */
                                                                         37
. . .
/* Find out the number of servers available */
MPI_Comm_remote_size ( multiple_server_comm, &num_servers );
/* Determine my color */
MPI_Comm_rank ( multiple_server_comm, &rank );
color = rank % num_servers;
/* Split the intercommunicator */
MPI_Comm_split ( multiple_server_comm, color, rank,
                 &single_server_comm );
```

**Unofficial Draft for Comment Only** 

1

2

3

4

5

6

7

8

9

10

11 12

13

14

15

16

17

18

19

2021

22

23

24

2526

27

28

2930

31

32

33

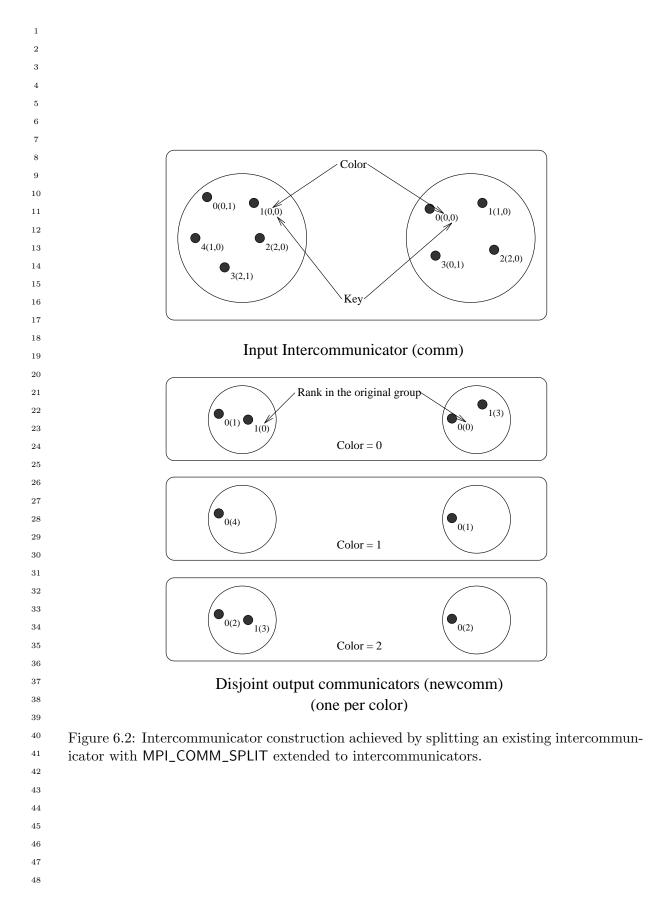
38 39

40 41 42

43 44

4546

47



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
                                                                                       39
```

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:

#### Unofficial Draft for Comment Only

1

40

41

42

43

44

45

46

47

1 MPI\_COMM\_TYPE\_SHARED — this type splits the communicator into subcommunicators, 2 each of which can create a shared memory region. 3 Advice to implementors. Implementations can define their own types, or use the 4 info argument, to assist in creating communicators that help expose platform-specific 5information to the application. (End of advice to implementors.) 6 7 8 Communicator Destructors 9 6.4.3 10 1112MPI\_COMM\_FREE(comm) 13 INOUT communicator to be destroyed (handle) comm 1415int MPI\_Comm\_free(MPI\_Comm \*comm) 1617 MPI\_Comm\_free(comm, ierror) BIND(C) 18 TYPE(MPI\_Comm), INTENT(INOUT) :: comm 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_COMM\_FREE(COMM, IERROR) 21INTEGER COMM, IERROR 2223This collective operation marks the communication object for deallocation. The handle  $^{24}$ is set to MPI\_COMM\_NULL. Any pending operations that use this communicator will com-25plete normally; the object is actually deallocated only if there are no other active references 26to it. This call applies to intra- and inter-communicators. The delete callback functions for 27all cached attributes (see Section 6.7) are called in arbitrary order. 2829Advice to implementors. A reference-count mechanism may be used: the reference 30 count is incremented by each call to MPI\_COMM\_DUP or MPI\_COMM\_IDUP, and 31decremented by each call to MPI\_COMM\_FREE. The object is ultimately deallocated 32 when the count reaches zero. 33 Though collective, it is anticipated that this operation will normally be implemented 34 to be local, though a debugging version of an MPI library might choose to synchronize. 35(End of advice to implementors.) 36 37 38 Communicator Info 6.4.4 39 40Hints specified via info (see Chapter 9) allow a user to provide information to direct opti-41 mization. Providing hints may enable an implementation to deliver increased performance 42or minimize use of system resources. However, hints do not change the semantics of any MPI 43interfaces. In other words, an implementation is free to ignore all hints. Hints are specified 44on a per communicator basis, in MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO, 45MPI\_COMM\_SPLIT\_TYPE, MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, and 46MPI\_DIST\_GRAPH\_CREATE, via the opaque info object. When an info object that speci-47fies a subset of valid hints is passed to MPI\_COMM\_SET\_INFO, there will be no effect on

<sup>48</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

Unofficial Draft for Comment Only

 $^{24}$ 

# <sup>1</sup> MPI\_COMM\_GET\_INFO(COMM, INFO\_USED, IERROR) <sup>2</sup> INTEGER COMM\_INEO\_USED\_IERROR

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.)

<sup>16</sup> 17 6.5 Motivating Examples

```
^{18}_{19} 6.5.1 Current Practice #1
```

3

4

5

6

7

8 9

10

11

12

13

14 15

35

```
19
20
     Example #1a:
21
         int main(int argc, char *argv[])
22
         ſ
23
           int me, size;
24
           . . .
25
           MPI_Init ( &argc, &argv );
26
           MPI_Comm_rank (MPI_COMM_WORLD, &me);
27
           MPI_Comm_size (MPI_COMM_WORLD, &size);
28
29
           (void)printf ("Process %d size %d\n", me, size);
30
           . . .
31
           MPI_Finalize();
32
           return 0;
33
        }
34
```

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

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

\frac{40}{10} int main(int args, char targy[])
```

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

```
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;
}
Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.
6.5.2 Current Practice #2</pre>
```

```
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)
  {
      /* 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;

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

22

23

 $^{24}$ 

25 26

27

28 29

30

 $^{31}$ 

32

33

34 35

36 37

38

39

40

 $41 \\ 42$ 

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

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

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

```
1
     MPI_Status status;
                                                                                      \mathbf{2}
     MPI_Wait(& handle -> isend_handle, &status);
                                                                                      3
     MPI_Wait(& handle -> irecv_handle, &status);
   }
                                                                                      4
                                                                                      5
User object clean-up code:
                                                                                      6
                                                                                      7
   void uninit_user_lib(user_lib_t *handle)
                                                                                      8
   {
                                                                                      9
     MPI_Comm_free(&(handle -> comm));
                                                                                      10
     free(handle);
                                                                                      11
   }
                                                                                      12
                                                                                      13
6.5.6 Library Example \#2
                                                                                      14
                                                                                      15
The main program:
                                                                                      16
   int main(int argc, char *argv[])
                                                                                      17
   ſ
                                                                                      18
     int ma, mb;
                                                                                      19
     MPI_Group MPI_GROUP_WORLD, group_a, group_b;
                                                                                      20
     MPI_Comm comm_a, comm_b;
                                                                                      21
                                                                                      22
     static int list_a[] = \{0, 1\};
                                                                                      23
#if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
                                                                                      24
     static int list_b[] = {0, 2,3};
                                                                                      25
#else/* EXAMPLE_2A */
                                                                                      26
     static int list_b[] = {0, 2};
                                                                                      27
#endif
                                                                                      28
     int size_list_a = sizeof(list_a)/sizeof(int);
                                                                                      29
     int size_list_b = sizeof(list_b)/sizeof(int);
                                                                                      30
                                                                                      31
     . . .
                                                                                      32
     MPI_Init(&argc, &argv);
                                                                                      33
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
                                                                                      34
                                                                                      35
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
                                                                                      36
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);
                                                                                      37
                                                                                      38
     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
                                                                                      39
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
                                                                                      40
                                                                                      41
     if(comm_a != MPI_COMM_NULL)
                                                                                      42
        MPI_Comm_rank(comm_a, &ma);
                                                                                      43
     if(comm_b != MPI_COMM_NULL)
                                                                                      44
        MPI_Comm_rank(comm_b, &mb);
                                                                                      45
                                                                                      46
     if(comm_a != MPI_COMM_NULL)
                                                                                      47
        lib_call(comm_a);
                                                                                      48
```

```
1
\mathbf{2}
           if(comm_b != MPI_COMM_NULL)
3
           {
4
              lib_call(comm_b);
5
              lib_call(comm_b);
6
           }
7
8
           if(comm_a != MPI_COMM_NULL)
9
              MPI_Comm_free(&comm_a);
10
           if(comm_b != MPI_COMM_NULL)
11
              MPI_Comm_free(&comm_b);
12
           MPI_Group_free(&group_a);
13
           MPI_Group_free(&group_b);
14
           MPI_Group_free(&MPI_GROUP_WORLD);
15
           MPI_Finalize();
16
           return 0;
17
         }
18
     The library:
19
         void lib_call(MPI_Comm comm)
20
         {
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
           {
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
42
      The above example is really three examples, depending on whether or not one includes rank
43
      3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates
^{44}
      that, despite contexts, subsequent calls to lib_call with the same context need not be safe
45
```

from one another (colloquially, "back-masking"). Safety is realized if the MPI\_Barrier is
 added. What this demonstrates is that libraries have to be written carefully, even with
 contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from
 back masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no backmasking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [58]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that backmasking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-topoint operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 6.9.

## 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 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

#### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10 11

12

13

14 15 16

17 18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

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	• 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	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).
23 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, <b>group</b> describes the remote group, and <b>source</b> is the rank of the process in the local group. For intra-communicators, <b>group</b> is the communicator group (remote=local), <b>source</b> is the rank of the process in this group, and <b>send context</b> and <b>receive context</b> 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 $\mathcal{P}$ .
46 47	• $\mathbf{C}_{\mathcal{P}}$ .source is rank of <b>P</b> in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$ .source is rank of <b>Q</b> in $\mathcal{Q}$ .
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.
	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

```
1
     MPI_CONGRUENT and MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result
\mathbf{2}
     because either the local or remote groups were similar but not identical.
3
         The following accessors provide consistent access to the remote group of an inter-
4
     communicator:
5
         The following are all local operations.
6
7
     MPI_COMM_REMOTE_SIZE(comm, size)
8
9
       IN
                 comm
                                             inter-communicator (handle)
10
       OUT
                                             number of processes in the remote group of comm
                 size
11
                                             (integer)
12
13
     int MPI_Comm_remote_size(MPI_Comm comm, int *size)
14
15
     MPI_Comm_remote_size(comm, size, ierror) BIND(C)
16
          TYPE(MPI_Comm), INTENT(IN) ::
                                             comm
17
          INTEGER, INTENT(OUT) :: size
18
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
19
     MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
20
          INTEGER COMM, SIZE, IERROR
21
22
23
     MPI_COMM_REMOTE_GROUP(comm, group)
^{24}
25
       IN
                 comm
                                             inter-communicator (handle)
26
       OUT
                 group
                                             remote group corresponding to comm (handle)
27
28
     int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
29
30
     MPI_Comm_remote_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
34
     MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
35
          INTEGER COMM, GROUP, IERROR
36
37
38
           Rationale.
                        Symmetric access to both the local and remote groups of an inter-
39
           communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE
40
          have been provided. (End of rationale.)
41
42
            Inter-communicator Operations
     6.6.2
43
     This section introduces four blocking inter-communicator operations.
44
     MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-com-
45
     municator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merg-
46
47
     ing the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP
48
```

and MPI\_COMM\_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multithreaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.)

The function MPI\_INTERCOMM\_CREATE can be used to create an inter-communicator 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)	31
IN	local_leader	rank of local group leader in local_comm (integer)	32 33
IN	peer_comm	"peer" communicator; significant only at the local_leader (handle)	34 35
IN	remote_leader	rank of remote group leader in <code>peer_comm</code> ; significant only at the <code>local_leader</code> (integer)	36 37
IN	tag	"safe" tag (integer)	38 39
OUT	newintercomm	new inter-communicator (handle)	40
			41
int MPI_I	ntercomm_create(MPI_Comm	<pre>local_comm, int local_leader,</pre>	42
	MPI_Comm peer_comm,	int remote_leader, int tag,	43
	MPI_Comm *newinterco	mm)	44
MPI_Inter		<pre>local_leader, peer_comm, remote_leader,</pre>	$45 \\ 46$
יייייייי	tag, newintercomm, i		47
IYPE(	MPI_Comm), INTENT(IN) ::	local_comm, peer_comm	48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11 12

13

14 15

16

17

18

19 20

21

22

23 24

25 26

27 28

29

```
1
          INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
\mathbf{2}
          TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
3
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
5
                    TAG, NEWINTERCOMM, IERROR)
6
          INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG.
7
          NEWINTERCOMM, IERROR
8
9
     This call creates an inter-communicator. It is collective over the union of the local and
10
     remote groups. Processes should provide identical local_comm and local_leader arguments
11
     within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.
12
13
     MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)
14
15
       IN
                 intercomm
                                            Inter-Communicator (handle)
16
       IN
                 high
                                             (logical)
17
       OUT
                 newintracomm
                                            new intra-communicator (handle)
18
19
20
     int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
21
                    MPI_Comm *newintracomm)
22
     MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) BIND(C)
23
          TYPE(MPI_Comm), INTENT(IN) :: intercomm
^{24}
          LOGICAL, INTENT(IN) :: high
25
          TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
26
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
27
28
     MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
29
          INTEGER INTERCOMM, NEWINTRACOMM, IERROR
30
          LOGICAL HIGH
^{31}
     This function creates an intra-communicator from the union of the two groups that are
32
     associated with intercomm. All processes should provide the same high value within each
33
     of the two groups. If processes in one group provided the value high = false and processes
34
     in the other group provided the value high = true then the union orders the "low" group
35
     before the "high" group. If all processes provided the same high argument then the order
36
     of the union is arbitrary. This call is blocking and collective within the union of the two
37
     groups.
38
          The error handler on the new intercommunicator in each process is inherited from
39
     the communicator that contributes the local group. Note that this can result in different
40
     processes in the same communicator having different error handlers.
41
42
           Advice to implementors.
                                       The implementation of MPI_INTERCOMM_MERGE,
43
          MPI_COMM_FREE and MPI_COMM_DUP are similar to the implementation of
44
          MPI_INTERCOMM_CREATE, except that contexts private to the input inter-com-
45
          municator are used for communication between group leaders rather than contexts
46
          inside a bridge communicator. (End of advice to implementors.)
47
48
```

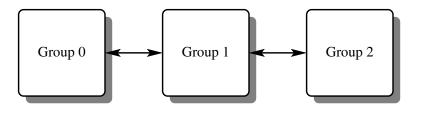


Figure 6.3: Three-group pipeline

#### 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.

```
16
int main(int argc, char *argv[])
                                                                                 17
{
                                                                                 18
 MPI_Comm
             myComm;
                            /* intra-communicator of local sub-group */
                                                                                 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. */
  ſ
                                                                                 48
```

1 2

9

10 11

12

13

14

```
1
2
3
                           Group 0
4
                                                                Group 2
                                              Group 1
5
6
7
                                   Figure 6.4: Three-group ring
8
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
```

```
2
  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
                                                                                 4
  /* Build inter-communicators. Tags are hard-coded. */
  if (membershipKey == 0)
  {
                 /* Group 0 communicates with groups 1 and 2. */
                                                                                 7
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                                                                                 9
                           1, &myFirstComm);
                                                                                 10
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                                                                                 11
                          2, &mySecondComm);
  }
                                                                                 12
  else if (membershipKey == 1)
                                                                                 13
                                                                                 14
  {
            /* Group 1 communicates with groups 0 and 2. */
                                                                                 15
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                                                                                 16
                           1, &myFirstComm);
                                                                                 17
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                                                                                 18
                          12, &mySecondComm);
                                                                                 19
  }
  else if (membershipKey == 2)
                                                                                 20
                                                                                 21
           /* Group 2 communicates with groups 0 and 1. */
  {
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                                                                                 22
                                                                                 23
                          2, &myFirstComm);
                                                                                 24
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                                                                                 25
                          12, &mySecondComm);
                                                                                 26
  }
                                                                                 27
  /* Do some work ... */
                                                                                 28
                                                                                 29
  /* Then free communicators before terminating... */
                                                                                 30
                                                                                 31
  MPI_Comm_free(&myFirstComm);
                                                                                 32
  MPI_Comm_free(&mySecondComm);
                                                                                 33
  MPI_Comm_free(&myComm);
                                                                                 34
  MPI_Finalize();
  return 0;
                                                                                 35
}
                                                                                 36
                                                                                 37
```

#### 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

3

5 6

38

39 40

41

42

43 44

4546

• be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI\_COMM\_SELF is a suitable choice for posting process-local attributes, via this attributing-caching mechanism. (*End of advice to users.*)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (End of rationale.)

One difficulty is the potential for size differences between Fortran integers and C pointers. 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 $comm\_keyval$ (function)
OUT	comm_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
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
10
     The C callback functions are:
11
     typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
12
                   void *extra_state, void *attribute_val_in,
13
                   void *attribute_val_out, int *flag);
14
15
     and
     typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
16
17
                   void *attribute_val, void *extra_state);
18
     which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
19
     With the mpi_f08 module, the Fortran callback functions are:
20
21
     ABSTRACT INTERFACE
22
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
23
       attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
24
           TYPE(MPI_Comm) :: oldcomm
25
           INTEGER :: comm_keyval, ierror
26
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
27
           attribute_val_out
28
           LOGICAL :: flag
29
30
     and
     ABSTRACT INTERFACE
31
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
32
       attribute_val, extra_state, ierror) BIND(C)
33
34
           TYPE(MPI_Comm) :: comm
           INTEGER :: comm_keyval, ierror
35
            INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
36
37
     With the mpi module and mpif.h, the Fortran callback functions are:
38
39
     SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
40
                   ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
41
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
42
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
43
              ATTRIBUTE_VAL_OUT
44
         LOGICAL FLAG
45
46
     and
47
     SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                   EXTRA_STATE, IERROR)
48
```

INTEGER	COMM,	COMM	_KEYVAL	, IERF	ROR			
INTEGER	(KIND=M	IPI_AI	DRESS_I	KIND)	ATTRIBUTE_	_VAL,	EXTRA_	STATE

The comm\_copy\_attr\_fn function is invoked when a communicator is duplicated by MPI\_COMM\_DUP or MPI\_COMM\_IDUP. comm\_copy\_attr\_fn should be of type MPI\_Comm\_copy\_attr\_function. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns flag = 0 or .FALSE, then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is set to the value returned in attribute\_val\_out. The function returns MPI\_SUCCESS on success and an error code on failure (in which case MPI\_COMM\_DUP or MPI\_COMM\_IDUP will fail).

The argument comm\_copy\_attr\_fn may be specified as MPI\_COMM\_NULL\_COPY\_FN 12or 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 the 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.

Advice to users. Even though both formal arguments attribute\_val\_in and attribute\_val\_out are of type void \*, their usage differs. The C copy function is passed by MPI in attribute\_val\_in the *value* of the attribute, and in attribute\_val\_out the address of the attribute, so as to allow the function to return the (new) attribute value. The use of type void \* for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (End of advice to users.)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (End of advice to implementors.)

Analogous to comm\_copy\_attr\_fn is a callback deletion function, defined as follows. The comm\_delete\_attr\_fn function is invoked when a communicator is deleted by MPI\_COMM\_FREE or when a call is made explicitly to MPI\_COMM\_DELETE\_ATTR. comm\_delete\_attr\_fn should be of type MPI\_Comm\_delete\_attr\_function.

This function is called by MPI\_COMM\_FREE, MPI\_COMM\_DELETE\_ATTR, and MPI\_COMM\_SET\_ATTR to do whatever is needed to remove an attribute. The function returns MPI\_SUCCESS on success and an error code on failure (in which case MPI\_COMM\_FREE will fail).

The argument comm\_delete\_attr\_fn may be specified as MPI\_COMM\_NULL\_DELETE\_FN 44from either C or Fortran. MPI\_COMM\_NULL\_DELETE\_FN is a function that does nothing, 45other than returning MPI\_SUCCESS. MPI\_COMM\_NULL\_DELETE\_FN replaces 46MPI\_NULL\_DELETE\_FN, whose use is deprecated. 47

5

6  $\overline{7}$ 

8 9

10

11

13

14

1516

17

18

1920

21

22

23

 $^{24}$ 

2526

27

28

29

30

31 32

33

34

35

36

37

38

39

40 41

42

43

1	If an attribute copy function or attribute delete function returns other than
2	MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE),
3	is erroneous.
4	The special key value MPI_KEYVAL_INVALID is never returned by
5	MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key
6	values.
7	
8	Advice to implementors. The predefined Fortran functions
9	MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
10	MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and
11	the mpi_f08 module with the same name, but with different interfaces. Each function
12	can coexist twice with the same name in the same MPI library, one routine as an
13	implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other
14	routine within mpi_f08 declared with CONTAINS. These routines have different link
15	names, which are also different to the link names used for the routines used in C.
16	(End of advice to implementors.)
17	
18	Advice to users. Callbacks, including the predefined Fortran functions
19	MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
20	MPI_COMM_NULL_DELETE_FN should not be passed from one application routine
21	that uses the mpi_f08 module to another application routine that uses the mpi module
22	or mpif.h, and vice versa, see also the advice to users on page 654. (End of advice to
23	users.)
24	
25	
26	MPI_COMM_FREE_KEYVAL(comm_keyval)
27 28	INOUT comm_keyval key value (integer)
28	
30	<pre>int MPI_Comm_free_keyval(int *comm_keyval)</pre>
31	Int MILComm_IIee_keyval(Int *Comm_keyval)
32	<pre>MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)</pre>
33	<pre>INTEGER, INTENT(INOUT) :: comm_keyval</pre>
34	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35	MDI COMM EDEE VEVUAI (COMM VEVUAI TEDDOD)
36	MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR) INTEGER COMM_KEYVAL, IERROR
37	INTEGEN COPHT_NEIVAL, IEMUON
38	Frees an extant attribute key. This function sets the value of keyval to
39	MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use,
40	because the actual free does not transpire until after all references (in other communicators
41	on the process) to the key have been freed. These references need to be explicitly freed by the
42	program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,
43	or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
43	
	communicator.
45	communicator.
45 46	communicator.
46	communicator.
	communicator.

MPI COM	IM_SET_ATTR(comm, comm.	kevval. attribute. val)	1		
INOUT	comm	communicator from which attribute will be attached	2		
		(handle)	3 4		
IN	comm_keyval	key value (integer)	5		
IN	attribute_val	attribute value	6		
			7		
int MPI_(	Comm_set_attr(MPI_Comm co	mm, int comm_keyval, void *attribute_val)	8 9		
MPI_Comm_	_set_attr(comm, comm_keyv	val, attribute_val, ierror) BIND(C)	10		
TYPE(MPI_Comm), INTENT(IN) :: comm					
	GER, INTENT(IN) :: comm_	•	12		
	GER(KIND=MPI_ADDRESS_KIND GER, OPTIONAL, INTENT(OUT		13		
			14 15		
	-	VAL, ATTRIBUTE_VAL, IERROR)	16		
	GER COMM, COMM_KEYVAL, IE GER(KIND=MPI_ADDRESS_KIND		17		
			18		
	-	attribute value attribute_val for subsequent retrieval due is already present, then the outcome is as if	19		
U		called to delete the previous value (and the callback	20 21		
		uted), and a new value was next stored. The call	21		
		ue keyval; in particular MPI_KEYVAL_INVALID is an	23		
erroneous	key value. The call will fail if	the <code>comm_delete_attr_fn</code> function returned an error	24		
code other	$t  an MPI_SUCCESS.$		25		
			26		
MPI_COM	IM_GET_ATTR(comm, comm	_keyval, attribute_val, flag)	27 28		
IN	comm	communicator to which the attribute is attached (han-	29		
		dle)	30		
IN	comm_keyval	key value (integer)	31		
OUT	attribute_val	attribute value, unless $flag = false$	32		
OUT	flag	false if no attribute is associated with the key (logical)	33		
001	nag	Taise if no attribute is associated with the key (logical)	34 35		
int MPT (	Comm get attr(MPI Comm co	mm, int comm_keyval, void *attribute_val,	36		
1110 111 1_1	int *flag)		37		
MDT Comm	ant attractory comm leaves	al attribute wel flog ierror) DIND(C)	38		
	_get_attr(comm, comm_keyv (MPI_Comm), INTENT(IN) ::	ral, attribute_val, flag, ierror) BIND(C)	39		
	GER, INTENT(IN) :: comm_		40		
		)), INTENT(OUT) :: attribute_val	41 42		
LOGI	CAL, INTENT(OUT) :: flag	5	42		
INTEG	GER, OPTIONAL, INTENT(OUT	C) :: ierror	44		
MPI_COMM	_GET_ATTR(COMM, COMM_KEYV	AL, ATTRIBUTE_VAL, FLAG, IERROR)	45		
	GER COMM, COMM_KEYVAL, IE		46		

INTEGER COMM, COMM\_KEYVAL, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL LOGICAL FLAG

**Unofficial Draft for Comment Only** 

1 2 3 4	keyval. Of attached o	n the other hand, th	by key. The call is erroneous if there is e call is correct if the key value exists, h in such case, the call returns flag = fa pneous key value.	out no attribute is			
5 6 7 8 9 10 11	the a loca a po will	Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_get_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.)					
12 13 14 15 16 17	void*	(*) avoids the messy	a formal parameter attribute_val of type type casting that would be needed if the r than void*. ( <i>End of rationale.</i> )	<b>`</b>			
18		IM_DELETE_ATTR(	omm comm keyval)				
19 20 21	INOUT	comm	communicator from which the attr dle)	ribute is deleted (han-			
22 23	IN	comm_keyval	key value (integer)				
24	int MPI_0	Comm_delete_attr(	PI_Comm comm, int comm_keyval)				
25 26 27 28 29	TYPE INTE	(MPI_Comm), INTEN GER, INTENT(IN) :					
30 31		_DELETE_ATTR(COMM GER COMM, COMM_KE	COMM_KEYVAL, IERROR) VAL, IERROR				
32 33 34 35 36 37 38 39 40 41	comm_del comm_del When MPI_COM invoked (i	ete_attr_fn specified ete_attr_fn function never a communicate IM_IDUP, all call-ba n arbitrary order).	e by key. This function invokes the attrib when the keyval was created. The call w eturns an error code other than MPI_SUC is replicated using the function MPI_CO ck copy functions for attributes that are whenever a communicator is deleted using a delete functions for attributes that are	ill fail if the CESS. DMM_DUP or e currently set are g the function			
42	6.7.3 W	indows					
43 44 45 46 47 48	The new f	unctions for caching	on windows are:				

MPI_WI	N_CREATE_KEYVAL(win_c	opy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)	1 2
IN	win_copy_attr_fn	copy callback function for win_keyval (function)	3
IN	win_delete_attr_fn	delete callback function for win_keyval (function)	4 5
OUT	win_keyval	key value for future access (integer)	6
IN	extra_state	extra state for callback functions	7 8
int MPI.	MPI_Win_delete_at	Win_copy_attr_function *win_copy_attr_fn, ttr_function *win_delete_attr_fn, void *extra_state)	9 10 11 12
	extra_state, ier	v_attr_fn, win_delete_attr_fn, win_keyval, ror) BIND(C) rr_function) :: win_copy_attr_fn	13 14 15
PRO		attr_function) :: win_delete_attr_fn	16 17
INT		(IND), INTENT(IN) :: extra_state	18 19
MPI_WIN	CREATE_KEYVAL(WIN_COPY EXTRA_STATE, IER	_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL, ROR)	20 21 22
	ERNAL WIN_COPY_ATTR_FN, EGER WIN_KEYVAL, IERROF	WIN_DELETE_ATTR_FN	22 23 24
INT	EGER(KIND=MPI_ADDRESS_K	(IND) EXTRA_STATE	25
MPI_WIN that does a simple-	$N_DUP_FN$ from either C s nothing other than return	in may be specified as MPI_WIN_NULL_COPY_FN or or Fortran. MPI_WIN_NULL_COPY_FN is a function ming flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is t sets flag = 1, returns the value of attribute_val_in in SUCCESS	26 27 28 29 30
The from eith	argument win_delete_attr_	_fn may be specified as MPI_WIN_NULL_DELETE_FN N_NULL_DELETE_FN is a function that does nothing,	31 32 33
The C ca	allback functions are:		34 35
typedef	void *extra_state	function(MPI_Win oldwin, int win_keyval, e, void *attribute_val_in,	36 37 38
	void *attribute_v	<pre>val_out, int *flag);</pre>	39
and typedef		r_function(MPI_Win win, int win_keyval,	40 41
	void *attribute_v	val, void *extra_state);	42
With the	e mpi_f08 module, the Fort	ran callback functions are:	43 44
ABSTRAC	I INTERFACE		45
		_function(oldwin, win_keyval, extra_state,	46
	bute_val_in, attribute_ YPE(MPI_Win) :: oldwin	val_out, flag, ierror) BIND(C)	47 48

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

0.11. 0110		210	
MPI_WIN_	_SET_ATTR(win, win_keyval, a	attribute_val)	1
INOUT	win	window to which attribute will be attached (handle)	2
IN	win_keyval	key value (integer)	3
	-		4
IN	attribute_val	attribute value	5
			6 7
int MPI_W	/in_set_attr(MPI_Win win,	<pre>int win_keyval, void *attribute_val)</pre>	8
MPI_Win_s	set_attr(win, win_keyval,	attribute_val, ierror) BIND(C)	9
	MPI_Win), INTENT(IN) ::		10
INTEG	ER, INTENT(IN) :: win_k	eyval	11
INTEG	ER(KIND=MPI_ADDRESS_KIND	), INTENT(IN) :: attribute_val	12
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	13
MPI_WIN_S	SET_ATTR(WIN, WIN_KEYVAL,	ATTRIBUTE_VAL, IERROR)	14
	ER WIN, WIN_KEYVAL, IERR	-	15
INTEG	ER(KIND=MPI_ADDRESS_KIND	) ATTRIBUTE_VAL	16
			17 18
			18
MPI_WIN_	_GET_ATTR(win, win_keyval,	attribute_val, flag)	20
IN	win	window to which the attribute is attached (handle)	21
IN	win_keyval	key value (integer)	22
	-	· · · · · · · · · · · · · · · · · · ·	23
OUT	attribute_val	attribute value, unless $flag = false$	24
OUT	flag	false if no attribute is associated with the key (logical)	25
			26
int MPI_W	0	<pre>int win_keyval, void *attribute_val,</pre>	27
	int *flag)		28 29
MPI_Win_g	get_attr(win, win_keyval,	attribute_val, flag, ierror) BIND(C)	30
TYPE(	MPI_Win), INTENT(IN) ::	win	31
INTEG	ER, INTENT(IN) :: win_k	eyval	32
		), INTENT(OUT) :: attribute_val	33
LOGIC	CAL, INTENT(OUT) :: flag		34

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

MPI\_WIN\_DELETE\_ATTR(win, win\_keyval) INOUT win window from which the attribute is deleted (handle) IN win\_keyval key value (integer)

int MPI\_Win\_delete\_attr(MPI\_Win win, int win\_keyval)

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

**Unofficial Draft for Comment Only** 

35 36

37

38

39

404142

43

44

45

4647

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

```
1
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
                                                                                     2
              int type_keyval, void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
                                                                                     4
and
                                                                                     5
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
                                                                                     6
              int type_keyval, void *attribute_val, void *extra_state);
With the mpi_f08 module, the Fortran callback functions are:
                                                                                     9
ABSTRACT INTERFACE
                                                                                     10
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                    11
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                    12
      TYPE(MPI_Datatype) :: oldtype
                                                                                    13
      INTEGER :: type_keyval, ierror
                                                                                    14
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                    15
      attribute_val_out
                                                                                     16
      LOGICAL :: flag
                                                                                     17
                                                                                     18
and
                                                                                     19
ABSTRACT INTERFACE
                                                                                    20
 SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                    21
  attribute_val, extra_state, ierror) BIND(C)
                                                                                    22
      TYPE(MPI_Datatype) :: datatype
                                                                                    23
      INTEGER :: type_keyval, ierror
                                                                                    24
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                    25
                                                                                    26
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                    27
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                    28
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                    29
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
                                                                                    30
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
                                                                                     31
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
                                                                                     32
    LOGICAL FLAG
                                                                                     33
                                                                                    34
and
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                    35
              EXTRA_STATE, IERROR)
                                                                                    36
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                    37
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                    38
                                                                                    39
   If an attribute copy function or attribute delete function returns other than
                                                                                     40
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
                                                                                    41
is erroneous.
                                                                                    42
                                                                                    43
                                                                                     44
MPI_TYPE_FREE_KEYVAL(type_keyval)
                                                                                     45
 INOUT
          type_keyval
                                     key value (integer)
                                                                                     46
                                                                                     47
                                                                                     48
int MPI_Type_free_keyval(int *type_keyval)
```

```
1
     MPI_Type_free_keyval(type_keyval, ierror) BIND(C)
\mathbf{2}
         INTEGER, INTENT(INOUT) :: type_keyval
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
5
          INTEGER TYPE_KEYVAL, IERROR
6
7
8
     MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
9
10
       INOUT
                datatype
                                            datatype to which attribute will be attached (handle)
11
                type_keyval
       IN
                                           key value (integer)
12
                attribute_val
                                            attribute value
       IN
13
14
15
     int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
16
                    void *attribute_val)
17
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         INTEGER, INTENT(IN) :: type_keyval
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
^{24}
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
25
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
26
27
28
     MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
29
                datatype
       IN
                                            datatype to which the attribute is attached (handle)
30
^{31}
       IN
                type_keyval
                                           key value (integer)
32
                attribute_val
       OUT
                                            attribute value, unless flag = false
33
       OUT
                flag
                                            false if no attribute is associated with the key (logical)
34
35
     int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void
36
                    *attribute_val, int *flag)
37
38
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
39
                    BIND(C)
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         INTEGER, INTENT(IN) :: type_keyval
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
43
         LOGICAL, INTENT(OUT) :: flag
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
46
47
          INTEGER DATATYPE, TYPE_KEYVAL, IERROR
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
48
```

```
1
    LOGICAL FLAG
                                                                                           \mathbf{2}
                                                                                           3
                                                                                           4
MPI_TYPE_DELETE_ATTR(datatype, type_keyval)
                                                                                           5
  INOUT
           datatype
                                        datatype from which the attribute is deleted (handle)
                                                                                           6
            type_keyval
  IN
                                        key value (integer)
                                                                                           7
                                                                                           8
                                                                                           9
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
                                                                                           10
MPI_Type_delete_attr(datatype, type_keyval, ierror) BIND(C)
                                                                                           11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                           12
    INTEGER, INTENT(IN) :: type_keyval
                                                                                           13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           14
                                                                                           15
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
                                                                                           16
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                           17
                                                                                           18
6.7.5
       Error Class for Invalid Keyval
                                                                                           19
Key values for attributes are system-allocated, by \mathsf{MPI}_{\mathsf{TYPE}}, \mathsf{COMM}, \mathsf{WIN}_{\mathsf{CREATE}} \mathsf{KEYVAL}_{21}^{20}
Only such values can be passed to the functions that use key values as input arguments.
                                                                                           22
In order to signal that an erroneous key value has been passed to one of these functions,
                                                                                           23
there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by
                                                                                           24
MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE,
                                                                                           25
MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR,
                                                                                           26
MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL,
                                                                                           27
MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DISCONNECT, and
                                                                                           28
MPI_COMM_FREE. The last three are included because keyval is an argument to the copy
                                                                                           29
and delete functions for attributes.
                                                                                           30
                                                                                           31
6.7.6 Attributes Example
                                                                                           32
                                                                                           33
     Advice to users.
                         This example shows how to write a collective communication
                                                                                           34
     operation that uses caching to be more efficient after the first call. The coding style
                                                                                           35
     assumes that MPI function results return only error statuses. (End of advice to users.)
                                                                                           36
                                                                                           37
   /* key for this module's stuff: */
                                                                                           38
   static int gop_key = MPI_KEYVAL_INVALID;
                                                                                           39
                                                                                           40
   typedef struct
                                                                                           41
   {
                                                                                           42
       int ref_count;
                                  /* reference count */
                                                                                           43
       /* other stuff, whatever else we want */
                                                                                           44
   } gop_stuff_type;
                                                                                           45
                                                                                           46
   void Efficient_Collective_Op (MPI_Comm comm, ...)
                                                                                           47
   {
                                                                                           48
```

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

```
void *extra)
{
  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)
ſ
  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.

```
39
MPI_COMM_SET_NAME (comm, comm_name)
                                                                                          40
  INOUT
           comm
                                       communicator whose identifier is to be set (handle)
                                                                                          41
  IN
                                       the character string which is remembered as the name
           comm_name
                                                                                         42
                                       (string)
                                                                                         43
                                                                                          44
int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
                                                                                          45
                                                                                          46
MPI_Comm_set_name(comm, comm_name, ierror) BIND(C)
                                                                                          47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          48
```

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	CHARA	CTER(LEN=*), INTENT(IN) :	: comm_name			
2		ER, OPTIONAL, INTENT(OUT)				
3	MPT COMM	SET_NAME(COMM, COMM_NAME,	TERROR)			
4		ER COMM, IERROR				
5		CTER*(*) COMM_NAME				
6 7						
8			ser to associate a name string with a communicator.			
9			MPI_COMM_SET_NAME will be saved inside the ller immediately after the call, or allocated on the			
10			ficant but trailing ones are not.			
11	/	0	(non-collective) operation, which only affects the			
12			process which made the MPI_COMM_SET_NAME			
13			time (or any) name be assigned to a communicator			
14		ocess where it exists.				
15						
16			M_SET_NAME is provided to help debug code, it			
17		0	to a communicator in all of the processes where it			
18	exists	s, to avoid confusion. (End of	advice to users.)			
19 20	The le	ength of the name which can	be stored is limited to the value of			
20		-	I MPI_MAX_OBJECT_NAME-1 in C to allow for the			
22			s longer than this will result in truncation of the			
23		_MAX_OBJECT_NAME must ha	-			
24						
25	Advice to users. Under circumstances of store exhaustion an attempt to put a name					
26			the value of MPI_MAX_OBJECT_NAME should be			
27			d on the name length, not a guarantee that setting			
28	names of less than this length will always succeed. (End of advice to users.)					
29	Advie	ce to implementors. Impleme	ntations which pre-allocate a fixed size space for a			
30 31		1 1 1	allocation as the value of MPI_MAX_OBJECT_NAME.			
32		0	ace for the name from the heap should still define			
33	MPI_	$MAX\_OBJECT\_NAME$ to be a r	elatively small value, since the user has to allocate			
34	space	e for a string of up to this size	e when calling MPI_COMM_GET_NAME. (End of			
35	advic	e to implementors.)				
36						
37						
38 39	MPI_COM	M_GET_NAME (comm, comm	_name, resultlen)			
40	IN	comm	communicator whose name is to be returned (handle)			
41 42	OUT	comm_name	the name previously stored on the communicator, or an empty string if no such name exists (string)			
43	OUT	resultlen	length of returned name (integer)			
44						
45 46	int MPI_C	comm_get_name(MPI_Comm com	m, char *comm_name, int *resultlen)			
47 48		get_name(comm, comm_name, MPI_Comm), INTENT(IN) ::	resultlen, ierror) BIND(C) comm			

```
CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
                                                                                            1
                                                                                            2
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
                                                                                            5
    INTEGER COMM, RESULTLEN, IERROR
                                                                                            6
    CHARACTER*(*) COMM NAME
                                                                                            7
                                                                                            8
    MPI_COMM_GET_NAME returns the last name which has previously been associated
                                                                                            9
with the given communicator. The name may be set and got from any language. The same
                                                                                            10
name will be returned independent of the language used. name should be allocated so that
                                                                                            11
it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters.
                                                                                            12
MPI_COMM_GET_NAME returns a copy of the set name in name.
                                                                                            13
    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
                                                                                            14
                                                                                            15
right with blank characters. The value of resultlen cannot be larger than
                                                                                            16
MPI_MAX_OBJECT_NAME.
                                                                                            17
    If the user has not associated a name with a communicator, or an error occurs.
                                                                                            18
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,
                                                                                            19
                                                                                            20
the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by
                                                                                            21
MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of
                                                                                            22
MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system
                                                                                            23
may have chosen to give a default name to a communicator does not prevent the user from
                                                                                            ^{24}
setting a name on the same communicator; doing this removes the old name and assigns
                                                                                            25
the new one.
                                                                                            26
     Rationale. We provide separate functions for setting and getting the name of a com-
                                                                                            27
     municator, rather than simply providing a predefined attribute key for the following
                                                                                            28
     reasons:
                                                                                            29
                                                                                            30
        • It is not, in general, possible to store a string as an attribute from Fortran.
                                                                                            ^{31}
        • It is not easy to set up the delete function for a string attribute unless it is known
                                                                                            32
          to have been allocated from the heap.
                                                                                            33
                                                                                            34
        • 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
                                                                                            35
          which we can easily eliminate.
                                                                                            36
                                                                                            37
        • The Fortran binding is not trivial to write (it will depend on details of the
                                                                                            38
          Fortran compilation system), and will not be portable. Therefore it should be in
                                                                                            39
          the library rather than in user code.
                                                                                            40
     (End of rationale.)
                                                                                            41
                                                                                            42
     Advice to users. The above definition means that it is safe simply to print the string
                                                                                            43
     returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was
                                                                                            44
     no name.
                                                                                            45
                                                                                            46
     Note that associating a name with a communicator has no effect on the semantics of
                                                                                            47
     an MPI program, and will (necessarily) increase the store requirement of the program,
                                                                                            48
     since the names must be saved. Therefore there is no requirement that users use these
```

functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (*End of advice to users.*)

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

```
MPI_TYPE_SET_NAME (datatype, type_name)
```

1

 $\mathbf{2}$ 

3

4

5

6

7 8 9

10

```
INOUT
                 datatype
                                            datatype whose identifier is to be set (handle)
11
       IN
                                            the character string which is remembered as the name
                 type_name
12
                                            (string)
13
14
15
     int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
16
     MPI_Type_set_name(datatype, type_name, ierror) BIND(C)
17
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
          CHARACTER(LEN=*), INTENT(IN) :: type_name
19
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
22
          INTEGER DATATYPE, IERROR
23
          CHARACTER*(*) TYPE_NAME
^{24}
25
26
     MPI_TYPE_GET_NAME (datatype, type_name, resultlen)
27
       IN
                 datatype
                                            datatype whose name is to be returned (handle)
28
       OUT
29
                 type_name
                                            the name previously stored on the datatype, or a empty
30
                                            string if no such name exists (string)
31
       OUT
                 resultlen
                                            length of returned name (integer)
32
33
     int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int
34
                    *resultlen)
35
36
     MPI_Type_get_name(datatype, type_name, resultlen, ierror) BIND(C)
37
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
          CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
39
          INTEGER, INTENT(OUT) :: resultlen
40
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
42
          INTEGER DATATYPE, RESULTLEN, IERROR
43
          CHARACTER*(*) TYPE_NAME
44
45
         Named predefined datatypes have the default names of the datatype name. For exam-
46
     ple, MPI_WCHAR has the default name of MPI_WCHAR.
47
         The following functions are used for setting and getting names of windows. The con-
```

#### $^{48}$ $\,$ stant MPI\_MAX\_OBJECT\_NAME also applies to these names.

MPI WI	N_SET_NAME (win, win_name		1		
	,	,	2		
INOUT		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,	ierror) BIND(C)	8 9		
	TYPE(MPI_Win), INTENT(IN) :: win				
	RACTER(LEN=*), INTENT(IN)		10		
	EGER, OPTIONAL, INTENT(OU		11		
MDT UTN			12 13		
	_SET_NAME(WIN, WIN_NAME, EGER WIN, IERROR	IERKUR)	13		
	RACTER*(*) WIN_NAME		15		
CIIA	RACIER*(*) WIN_NAME		16		
			17		
	N_GET_NAME (win, win_name	e resultion)	18		
	Ύ	, , ,	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	21		
		string if no such name exists (string)	22		
OUT	resultlen	length of returned name (integer)	23		
			24		
int MPI	_Win_get_name(MPI_Win win	, char *win_name, int *resultlen)	25		
MDT U	-	$(\alpha)$	26 27		
	_get_name(win, win_name, E(MPI_Win), INTENT(IN) ::	resultlen, ierror) BIND(C) win	28		
	RACTER(LEN=MPI_MAX_OBJECT		29		
	EGER, INTENT(OUT) :: res		30		
	EGER, OPTIONAL, INTENT(OU		31		
			32		
	_GET_NAME(WIN, WIN_NAME,		33		
	EGER WIN, RESULTLEN, IERR	UR	34		
CHA	RACTER*(*) WIN_NAME		35		
			36		
6.9 F	ormalizing the Loosely S	vnchronous Model	37		
0.9 1	Simalizing the Loosely S		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

#### Unofficial Draft for Comment Only

39

40

 $41 \\ 42$ 

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 32

## <sup>32</sup> Dynamic communicator allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in
 a subset of a group executing the same parallel procedure. Thus, processes that execute
 the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI\_COMM\_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI\_COMM\_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI\_ANY\_SOURCE).

#### 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, then communicator creation be properly coordinated.  $\mathbf{2}$ 

 $14 \\ 15$ 

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			
48			

## Chapter 7

# **Process Topologies**

#### 7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to intercommunicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal 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 [45]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [11, 12].

 $^{24}$ 

 $^{31}$ 

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. 24

<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.

28 29

30

 $\begin{array}{c} {
m coord} \ (0,0): \ \ {
m rank} \ 0 \ {
m coord} \ (0,1): \ \ {
m rank} \ 1 \ {
m coord} \ (1,0): \ \ {
m rank} \ 2 \end{array}$ 

coord (1,1):

31 32 33

34

35 36

37

38

## 7.3 Embedding in MPI

rank 3

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.

39 40 41

42

## 7.4 Overview of the Functions

<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>40</sup> MPI\_DIST\_GRAPH\_CREATE\_ADJACENT and MPI\_DIST\_GRAPH\_CREATE are used to cre-

 $\frac{1}{48}$  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 34 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. 37 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 gen-43 eral, not called by the user directly. However, together with the communicator manipulation 44functions presented in Chapter 6, they are sufficient to implement all other topology func-45tions. Section 7.5.8 outlines such an implementation. 46

The neighborhood collective communication routines MPI\_NEIGHBOR\_ALLGATHER, 47 MPI\_NEIGHBOR\_ALLGATHERV, MPI\_NEIGHBOR\_ALLTOALL, MPI\_NEIGHBOR\_ALLTOALLV, 48

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10

11

12

13

14

15

16

17

18

1920

21

22

23

 $^{24}$ 25

26

27

28

29

30

31

32

33

35

36

38

39

40

41

```
1
     and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topol-
\mathbf{2}
     ogy associated with the communicator. The nonblocking variants are
3
     MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,
4
     MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and
5
     MPI_INEIGHBOR_ALLTOALLW.
6
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. If ndims is zero then a zero-dimensional Cartesian topology is created.
45
     The call is erroneous if it specifies a grid that is larger than the group size or if ndims is
46
     negative.
47
48
```

CHAPTER 7. PROCESS TOPOLOGIES

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					
	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.			5		
				6		
n-unionsional topology.				7		
				8 9		
	MPI_DIMS_CREATE(nnodes, ndims, dims)					
	IN	nnodes	number of nodes in a grid (integer)	10 11		
	IN	ndims	number of Cartesian dimensions (integer)	12		
				13		
	INOUT	dims	integer array of size ndims specifying the number of			
			nodes in each dimension	15		
	<pre>int MPI_Dims_create(int nnodes, int ndims, int dims[])</pre>					
	MPI_Dims_create(nnodes, ndims, dims, ierror) BIND(C)			18		
	INTEGER, INTENT(IN) :: nnodes, ndims			19		
	INTEGER, INTENT(INOUT) :: dims(ndims)					
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			21		
	MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)					
	INTEGER NNODES, NDIMS, DIMS, IERROR			23		
				24 25		
	The entries in the array dims are set to describe a Cartesian grid with ndims dimensions					
	and a total of <b>nnodes</b> 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 <sup>27</sup>					

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 dims[i] are erroneous. An error will occur if nnodes is not a multiple of  $\prod dims[i]$ .

 $i,dims[i] \neq 0$ 

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI\_CART\_CREATE. MPI\_DIMS\_CREATE is local.

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

 $^{28}$ 

29 30

 $^{31}$ 

32

33

34

35

294

1

2 3 4

5

#### 7.5.3 Graph Constructor

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	<pre>const int edges[], int reorder, MPI_Comm *comm_graph)</pre>					
17	MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,					
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	MDT CDADU	CDEATE COMM OLD NNODES	TNDEY EDGEG DEODDED COMM CDADU			
25	MPI_GRAPH		, INDEX, EDGES, REORDER, COMM_GRAPH,			
26	IERROR)					
27	INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR					
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  $^{31}$ new 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.

46

#### 47 Example 7.2

48

#### Unofficial Draft for Comment Only

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

nnodes =	4
index =	2, 3, 4, 6
edges =	1, 3, 0, 3, 0,

 $\mathbf{2}$ 

Thus, in C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges[j], for  $0 \le j \le index[0] - 1$  and the list of neighbors of node i, i > 0, is stored in edges[j], index[i - 1]  $\le j \le index[i] - 1$ .

In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in 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

# Unofficial Draft for Comment Only

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  $\mathbf{5}$ specifies 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	IN	indegree	size of sources and source weights $\operatorname{arrays}$ (non-negative
26			integer)
27	IN	sources	ranks of processes for which the calling process is a
28			destination (array of non-negative integers)
29	IN	sourceweights	weights of the edges into the calling process (array of
30 31			non-negative integers)
32	IN	outdegree	size of destinations and destweights arrays (non-negative
33		outdegree	integer)
34			- /
35	IN	destinations	ranks of processes for which the calling process is a
36			source (array of non-negative integers)
37	IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
38 39	IN	info	hints on optimization and interpretation of weights
40	IIN	IIIO	(handle)
41	IN	reorder	the ranks may be reordered (true) or not (false) (logi-
42			cal)
43	OUT	comm_dist_graph	communicator with distributed graph topology (han-
44	001	comm_dist_graph	dle)
45			
46	int MDT T	)ist graph graata adjacant	(MPI Comm comm old int indogroo const
47	IIIC MPI_I		(MPI_Comm comm_old, int indegree, const int sourceweights[], int outdegree, const
48		Int Sources[], CONSt	THE SOULCEWEIGHTS[], THE OUCCEGIEE, CONSE

7.5.4

```
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,
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
                                                                                   19
                                                                                   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

#### Unofficial Draft for Comment Only

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. 1920MPI\_DIST\_GRAPH\_CREATE(comm\_old, n, sources, degrees, destinations, weights, info, re-21order, comm\_dist\_graph) 22 23IN 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)

**Unofficial Draft for Comment Only** 

```
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. 19Every 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 22precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the j-th such edge stored in destinations[degrees[0]+...+degrees[i-1]+j]. The weight of this edge is 23stored in weights [degrees[0]+...+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 27with the same source and destination nodes. Source and destination nodes must be pro-28cess ranks of comm\_old. Different processes may specify different numbers of source and 29destination nodes, as well as different source to destination edges. This allows a fully dis-30 tributed specification of the communication graph. Isolated processes (i.e., processes with  $^{31}$ no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed. 33

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

 $^{24}$ 

25

26

32

34

35

36

37

38

39

40

41

1 the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have the  $\mathbf{2}$ same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some but not 3 all processes of comm\_old. If the graph is weighted but n = 0, then MPI\_WEIGHTS\_EMPTY 4 or any arbitrary array may be passed to weights. Note that MPI\_UNWEIGHTED and  $\mathbf{5}$ MPI\_WEIGHTS\_EMPTY are not special weight values; rather they are special values for the 6 total array argument. In Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects  $\overline{7}$ like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 8 9 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 10 MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then 11 be indistinguishable from MPI\_UNWEIGHTED to the implementation. In this case 12MPI\_WEIGHTS\_EMPTY should be used instead. (End of advice to users.) 13 14Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be imple-15mented as NULL. (End of advice to implementors.) 1617 *Rationale.* To ensure backward compatibility, MPI\_UNWEIGHTED may still be imple-18 mented as NULL. See Annex B.1 on page 789. (End of rationale.) 1920The meaning of the weights argument can be influenced by the info argument. Info 21arguments can be used to guide the mapping; possible options include minimizing the 22 maximum number of edges between processes on different SMP nodes, or minimizing the 23sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it 24is valid for an MPI implementation not to do any reordering. An MPI implementation may 25specify more info key-value pairs. All processes must specify the same set of key-value info 26pairs. 2728 Advice to implementors. MPI implementations must document any additionally 29 supported key-value info pairs. MPI\_INFO\_NULL is always valid, and may indicate the 30 default creation of the distributed graph topology to the MPI library.  $^{31}$ An implementation does not explicitly need to construct the topology from its dis-32 tributed parts. However, all processes can construct the full topology from the dis-33 tributed specification and use this in a call to MPI\_GRAPH\_CREATE to create the 34 topology. This may serve as a reference implementation of the functionality, and 35 may be acceptable for small communicators. However, a scalable high-quality im-36 plementation would save the topology graph in a distributed way. (End of advice to 37 *implementors.*) 38 39 40 **Example 7.3** As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the 41 following adjacency matrix and unit edge weights: 4243

	process	neighbors
ſ	0	1, 3
	1	0
	2	3
	3	0, 2

With MPI\_DIST\_GRAPH\_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	$1,\!3$	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	$1,\!1,\!1,\!1,\!1,\!1,\!1$
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

In both cases above, the application could supply MPI\_UNWEIGHTED instead of 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:

Unofficial Draft for Comment Only

 $45 \\ 46$ 

```
1
     MPI_Comm comm_dist_graph;
\mathbf{2}
3
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4
5
     /* get x and y dimension */
6
     y=rank/P; x=rank%P;
7
8
     /* get my communication partners along x dimension */
9
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
10
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
11
12
     /* get my communication partners along y dimension */
13
     destinations[2] = P*((y+1))(x) + x; weights[2] = 2;
14
     destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
15
16
     /* get my communication partners along diagonals */
17
     destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
18
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
19
     destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
20
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
21
22
     sources[0] = rank;
23
     degrees[0] = 8;
^{24}
     MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
25
                             weights, MPI_INFO_NULL, 1, &comm_dist_graph);
26
27
           Topology Inquiry Functions
     7.5.5
28
     If a topology has been defined with one of the above functions, then the topology information
29
     can be looked up using inquiry functions. They all are local calls.
30
^{31}
32
     MPI_TOPO_TEST(comm, status)
33
       IN
                                           communicator (handle)
                 comm
34
35
       OUT
                                           topology type of communicator comm (state)
                 status
36
37
     int MPI_Topo_test(MPI_Comm comm, int *status)
38
     MPI_Topo_test(comm, status, ierror) BIND(C)
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         INTEGER, INTENT(OUT) :: status
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
44
         INTEGER COMM, STATUS, IERROR
45
         The function MPI_TOPO_TEST returns the type of topology that is assigned to a
46
47
     communicator.
         The output value status is one of the following:
48
```

MPI_GRA MPI_CAR MPI_DIST MPI_UND	T _GRAPH	graph topology Cartesian topology distributed graph topology no topology	1 2 3 4 5
			6
MPI_GRAF	HDIMS_GET(comm, nnodes,	nedges)	7
IN	comm	communicator for group with graph structure (handle)	8
OUT	nnodes	number of nodes in graph (integer) (same as number	9 10
001	modes	of processes in the group)	10
OUT	nedges	number of edges in graph (integer)	12
001	1104800	namber of edges in graph (messer)	13
int MPI_G	raphdims_get(MPI_Comm co	mm, int *nnodes, int *nedges)	14
			15
-	dims_get(comm, nnodes, n MPI_Comm), INTENT(IN) ::	-	16 17
	ER, INTENT(OUT) :: nnod		18
	ER, OPTIONAL, INTENT(OUT		19
МРТ СВАРН	DIMS_GET(COMM, NNODES, N	FDCFS IFRROR)	20
	ER COMM, NNODES, NEDGES,		21
			22
		and MPI_GRAPH_GET retrieve the graph-topology communicator by MPI_GRAPH_CREATE.	23 24
		_GRAPHDIMS_GET can be used to dimension the	25
	* 0	he following call to MPI_GRAPH_GET.	26
			27
MPI GRAP	'H_GET(comm, maxindex, ma	xedges index edges)	28
		,	29 30
IN	comm	communicator with graph structure (handle)	31
IN	maxindex	length of vector index in the calling program	32
		(integer)	33
IN	maxedges	length of vector <b>edges</b> in the calling program	34
		(integer)	35
OUT	index	array of integers containing the graph structure (for	36
		details see the definition of MPI_GRAPH_CREATE)	37 38
OUT	edges	array of integers containing the graph structure	39
			40
int MPI_G	<pre>int edges[])</pre>	<pre>int maxindex, int maxedges, int index[],</pre>	41
	-		42
-	-	edges, index, edges, ierror) BIND(C)	43
	MPI_Comm), INTENT(IN) :: ER, INTENT(IN) :: maxin		44 45
		dex, maxeages x(maxindex), edges(maxedges)	46
	ER, OPTIONAL, INTENT(OUT		47
	, , (000		48

```
304
                                                  CHAPTER 7. PROCESS TOPOLOGIES
1
     MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
\mathbf{2}
          INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
3
4
\mathbf{5}
     MPI_CARTDIM_GET(comm, ndims)
6
       IN
                                            communicator with Cartesian structure (handle)
                 comm
7
       OUT
8
                 ndims
                                             number of dimensions of the Cartesian structure (in-
9
                                             teger)
10
11
     int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
12
     MPI_Cartdim_get(comm, ndims, ierror) BIND(C)
13
          TYPE(MPI_Comm), INTENT(IN) :: comm
14
          INTEGER, INTENT(OUT) :: ndims
15
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
16
17
     MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
18
          INTEGER COMM, NDIMS, IERROR
19
         The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topol-
20
     ogy information that was associated with a communicator by MPI_CART_CREATE. If comm
21
     is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns
22
     ndims=0 and MPI_CART_GET will keep all output arguments unchanged.
23
24
25
     MPI_CART_GET(comm, maxdims, dims, periods, coords)
26
       IN
                                            communicator with Cartesian structure (handle)
                 comm
27
28
       IN
                 maxdims
                                            length of vectors dims, periods, and coords in the
29
                                            calling program (integer)
30
       OUT
                 dims
                                            number of processes for each Cartesian dimension (ar-
^{31}
                                            ray of integer)
32
       OUT
                 periods
                                             periodicity (true/false) for each Cartesian dimension
33
                                             (array of logical)
34
35
       OUT
                 coords
                                             coordinates of calling process in Cartesian structure
36
                                             (array of integer)
37
38
     int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
39
                    int coords[])
40
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) BIND(C)
41
          TYPE(MPI_Comm), INTENT(IN) :: comm
42
          INTEGER, INTENT(IN) :: maxdims
43
          INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
44
          LOGICAL, INTENT(OUT) :: periods(maxdims)
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
48
```

INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR			1	
LOGICA	LOGICAL PERIODS(*)		2	
			3	
			4	
MPI_CART	_RANK(comm, coords, rank)		5	
IN	comm	communicator with Cartesian structure (handle)	6 7	
IN	coords	integer array (of size ndims) specifying the Cartesian	8	
		coordinates of a process	9	
OUT	rank	rank of specified process (integer)	10	
			11	
int MPT Ca	ort rank(MPI Comm comm, o	const int coords[], int *rank)	12	
1110 111 1_00			13	
MPI_Cart_r	<pre>rank(comm, coords, rank,</pre>	ierror) BIND(C)	14	
	<pre>IPI_Comm), INTENT(IN) ::</pre>		15	
	ER, INTENT(IN) :: coords	s(*)	16	
	CR, INTENT(OUT) :: rank		17	
INTEGE	CR, OPTIONAL, INTENT(OUT)	:: ierror	18 19	
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)				
	ER COMM, COORDS(*), RANK,		20	

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)  $\geq$  dims(i), it is shifted back to the interval  $0 \leq \text{coords}(i) < \text{dims}(i)$  automatically. Out-of-range coordinates are erroneous for non-periodic 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	comm	communicator with Cartesian structure (handle)	34
IN	rank	rank of a process within group of $comm$ (integer)	35 36
IN	maxdims	length of vector coords in the calling program (inte-	37
		ger)	38
OUT	coords	integer array (of size $\texttt{ndims}$ ) containing the Cartesian	39
		coordinates of specified process (array of integers)	40 41
			41

int MPI\_Cart\_coords(MPI\_Comm comm, int rank, int maxdims, int coords[])

<pre>MPI_Cart_coords(comm, rank, maxdims, coords, ierror) BIND(C)</pre>	44
TYPE(MPI_Comm), INTENT(IN) :: comm	45
INTEGER, INTENT(IN) :: rank, maxdims	46
<pre>INTEGER, INTENT(OUT) :: coords(maxdims)</pre>	47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

Unofficial Draft for Comment Only

22 23

 $^{24}$ 

25

26

27 28

29

30

31 32

33

42

```
1
     MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
\mathbf{2}
          INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
3
         The inverse mapping, rank-to-coordinates translation is provided by
4
     MPI_CART_COORDS.
5
         If comm is associated with a zero-dimensional Cartesian topology,
6
     coords will be unchanged.
7
8
9
     MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)
10
       IN
                 comm
                                            communicator with graph topology (handle)
11
       IN
                 rank
                                            rank of process in group of comm (integer)
12
13
       OUT
                 nneighbors
                                            number of neighbors of specified process (integer)
14
15
     int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
16
17
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) BIND(C)
18
          TYPE(MPI_Comm), INTENT(IN) :: comm
19
          INTEGER, INTENT(IN) :: rank
          INTEGER, INTENT(OUT) :: nneighbors
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
23
          INTEGER COMM, RANK, NNEIGHBORS, IERROR
24
25
26
     MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
27
28
       IN
                                            communicator with graph topology (handle)
                 comm
29
       IN
                 rank
                                            rank of process in group of comm (integer)
30
                 maxneighbors
                                            size of array neighbors (integer)
       IN
^{31}
32
       OUT
                 neighbors
                                            ranks of processes that are neighbors to specified pro-
33
                                            cess (array of integer)
34
35
     int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
36
                    int neighbors[])
37
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) BIND(C)
38
          TYPE(MPI_Comm), INTENT(IN) :: comm
39
          INTEGER, INTENT(IN) :: rank, maxneighbors
40
          INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
44
          INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
45
          MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
46
47
     information for a graph topology. The returned count and array of neighbors for the queried
     rank will both include all neighbors and reflect the same edge ordering as was specified by
48
```

the original call to MPI\_GRAPH\_CREATE. Specifically, MPI\_GRAPH\_NEIGHBORS\_COUNT and MPI\_GRAPH\_NEIGHBORS will return values based on the original index and edges array passed to MPI\_GRAPH\_CREATE (assuming that index[-1] effectively equals zero):

- 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 arguments to  $\mathsf{MPI}\_\mathsf{GRAPH}\_\mathsf{CREATE}$  are:

nnodes =	4
index =	3,5,6,9
edges =	1, 1, 3, 0, 0, 3, 0, 2, 2

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.

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	
			-	ogy associated with it. The
-		hrough the thre	ee types of neigl	nbors and performs an app
permutati	on for each.			
C assum	e: each proces	a has stored	a real numbe	r A
	ct neighborhod			1 A.
	LL MPI_COMM_RA			
		•		neighbors, ierr)
	rm exchange pe		m, myrank, J,	neighbors, ieil)
-	01			neighbors(1), 0,
+			status, ierr)	0
	rm shuffle per		Status, ieii/	
-	-		1 MPT REAL	neighbors(2), 0,
+			status, ierr)	neignoois(2), 0,
	rm unshuffle p		status, ieii)	
-	-			neighbors(3), 0,
+			status, ierr)	•
	neignborb (2	, o, comi,	boards, icii)	
MPI_	DIST_GRAPH_N	EIGHBORS_CO	DUNT and MP	_DIST_GRAPH_NEIGHBO
vide adjao	ency information	n for a distribu	ted graph topol	ogy.
עם וא	GRAPH NEIG	BORS COUN	T(comm_indegr	ee, outdegree, weighted)
		1201020001		,
IN	comm			with distributed graph topolo
			dle)	
OUT	indegree		number of edges	s into this process (non-negat
			ger)	
OUT	outdegree		number of edges	s out of this process (non-neg
			teger)	i i i i i i i i i i i i i i i i i i i
	المعامة المعام			
OUT	weighted			WEIGHTED was supplied du
			ation, true other	rwise (logical)
int MPI_	•••	•		m, int *indegree,
	int *outo	legree, int *	weighted)	
MPT Dist	graph neighbo	rs count (com	m. indegree	outdegree, weighted, i
	_graph_nergnbc BIND(C)	1.5_00 and (00 m	, 111405100,	sataobroo, worknood, r
	DIND(0)			

INTE LOGI		egree, outdegree ghted	1 2 3 4
INTE	_GRAPH_NEIGHBORS_COUNT(C GER COMM, INDEGREE, OUTD CAL WEIGHTED	COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) EGREE, IERROR	5 6 7 8 9
MPI_DIST	GRAPH_NEIGHBORS(com destinations, destweight	m, maxindegree, sources, sourceweights, maxoutdegree, ts)	10 11 12
IN	comm	communicator with distributed graph topology (han-dle)	13 14
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)	15 16 17
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)	18 19
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)	20 21
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer) $% \left( {{\left[ {{{\rm{max}}} \right]}_{{\rm{max}}}}_{{\rm{max}}}} \right)$	22 23 24
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	25 26
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	27 28 29
int MPI_	010	<pre>_Comm comm, int maxindegree, int sources[], , int maxoutdegree, int destinations[],</pre>	30 31 32 33
TYPE INTE INTE dest INTE		ndegree, maxoutdegree rces(maxindegree), , destweights(*)	34 35 36 37 38 39 40 41
INTE	MAXOUTDEGREE, DESTI	AXINDEGREE, SOURCES, SOURCEWEIGHTS, INATIONS, DESTWEIGHTS, IERROR) OURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, GHTS(*), IERROR	42 43 44 45 46
		er of edges into and out of the process returned by JNT are the total number of such edges given in the	47 48

1 call to MPI\_DIST\_GRAPH\_CREATE\_ADJACENT or MPI\_DIST\_GRAPH\_CREATE (poten- $\mathbf{2}$ tially by processes other than the calling process in the case of 3 MPI\_DIST\_GRAPH\_CREATE). Multiply defined edges are all counted and returned by 4 MPI\_DIST\_GRAPH\_NEIGHBORS in some order. If MPI\_UNWEIGHTED is supplied for  $\mathbf{5}$ sourceweights or destweights or both, or if MPI\_UNWEIGHTED was supplied during the con-6 struction of the graph then no weight information is returned in that array or those arrays. 7If the communicator was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT then for 8 each rank in comm, the order of the values in sources and destinations is identical to the in-9 put that was used by the process with the same rank in **comm\_old** in the creation call. If the 10 communicator was created with MPI\_DIST\_GRAPH\_CREATE then the only requirement on 11the order of values in **sources** and **destinations** is that two calls to the routine with same input 12argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is 13smaller than the numbers returned by MPI\_DIST\_GRAPH\_NEIGHBOR\_COUNT, then only 14the first part of the full list is returned. 15

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.)

<sup>22</sup> 7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI\_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI\_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI\_CART\_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI\_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

31 32

16

17

18 19

20 21

```
MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
```

33			, Talk_Source, Talk_dest)
34	IN	comm	communicator with Cartesian structure (handle)
35	IN	direction	coordinate dimension of shift (integer)
36 37	IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
38 39	OUT	rank_source	rank of source process (integer)
40	OUT	rank_dest	rank of destination process (integer)
41			
42	int MPI_Ca	art_shift(MPI_Comm comm,	int direction, int disp,
43		int *rank_source, in	t *rank_dest)
44			
45	MPI_Cart_s	shift(comm, direction, di	isp, rank_source, rank_dest, ierror)
46		BIND(C)	
47	TYPE(N	<pre>MPI_Comm), INTENT(IN) ::</pre>	comm
48	INTEG	ER, INTENT(IN) :: direct	cion, disp

INTEGER, INTENT(OUT) :: rank_source, rank_dest INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
	3
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)	4
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR	5
The direction argument indicates the coordinate dimension to be traversed by the shift.	6
The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.	7
Depending on the periodicity of the Cartesian group in the specified coordinate direc-	8
tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case	9
of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest,	10
indicating that the source or the destination for the shift is out of range.	11
It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or	12
greater than or equal to the number of dimensions in the Cartesian communicator. This	13
implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with	14
a zero-dimensional Cartesian topology.	15
a zero-dimensional Cartesian topology.	16
Example 7.7	17
The communicator, comm, has a two-dimensional, periodic, Cartesian topology associ-	18
ated with it. A two-dimensional array of REALs is stored one element per process, in variable	19
A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column)	20
by i steps.	21
	22
	23
C find process rank	24
CALL MPI_COMM_RANK(comm, rank, ierr)	25
C find Cartesian coordinates	26
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)	27
C compute shift source and destination	28
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)	29
C skew array	30
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,	31
+ status, ierr)	32
	33
Advice to users. In Fortran, the dimension indicated by $DIRECTION = i$ has $DIMS(i+1)$	34
nodes, where DIMS is the array that was used to create the grid. In C, the dimension	35
indicated by direction = i is the dimension specified by dims[i]. (End of advice to users.)	36
	37
	38
	39
	40
	41
	42
	43
	44
	45

	312		CHAPTER 7.	PROCESS TOPOLOGIES	
1 2 3	7.5.7 Pa	rtitioning of Cartesian Struct	ures		
4	MPI_CAR	T_SUB(comm, remain_dims, no	ewcomm)		
5 6	IN	comm	communicator with Ca	rtesian structure (handle)	
7 8 9	IN	remain_dims	v	ain_dims specifies whether the in the subgrid (true) or is drop- ector)	
10 11 12	OUT	newcomm	communicator contain the calling process (ha	ing the subgrid that includes ndle)	
13 14	int MPI_0	Cart_sub(MPI_Comm comm, c	onst int remain_dim	s[], MPI_Comm *newcomm)	
15 16 17 18 19	TYPE LOGI TYPE	_sub(comm, remain_dims, n (MPI_Comm), INTENT(IN) :: CAL, INTENT(IN) :: remain (MPI_Comm), INTENT(OUT) : GER, OPTIONAL, INTENT(OUT)	comm n_dims(*) : newcomm	D(C)	
20 21 22 23	MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR) INTEGER COMM, NEWCOMM, IERROR LOGICAL REMAIN_DIMS(*)				
24 25 26 27 28 29 30 31	MPI_CAR form lower with the a comm is a associated	Cartesian topology has been T_SUB can be used to partit r-dimensional Cartesian subgri associated subgrid Cartesian to lready associated with a zero with a zero-dimensional Cart IM_SPLIT.)	ion the communicator ds, and to build for eac copology. If all entries -dimensional Cartesian	group into subgroups that ch subgroup a communicator in remain_dims are false or topology then newcomm is	
32 33 34		7.8 ne that MPI_CART_CREATE ims = (true, false, true)		ned a $(2 \times 3 \times 4)$ grid. Let	
35 36	MPI.	_CART_SUB(comm, remain_di	ns, comm_new),		
37 38 39 40 41	ogy. If re remain_dir	e three communicators each main_dims = (false, false ns, comm_new) will create sin in a one-dimensional Cartesia	e, true) then the call x non-overlapping com	to MPI_CART_SUB(comm,	
42 43	7.5.8 Lo	w-Level Topology Functions			
43 44 45 46	topology f	dditional functions introduced functions. In general they will gadditional virtual topology ca	not be called by the u	ser directly, unless he or she	

<sup>47</sup> calls are both local.

MPI_CART	_MAP(comm, ndims, dims, pe	eriods, newrank)	1
IN	comm	input communicator (handle)	2 3
IN	ndims	number of dimensions of Cartesian structure (integer)	3 4
IN	dims	integer array of size ndims specifying the number of processes in each coordinate direction	5 6
IN	periods	logical array of size <b>ndims</b> specifying the periodicity specification in each coordinate direction	7 8 9
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)	10 11 12 13
int MPI_C	art_map(MPI_Comm comm, in int periods[], int *:	nt ndims, const int dims[], const newrank)	14 15
TYPE() INTEG LOGIC INTEG	<pre>map(comm, ndims, dims, pe MPI_Comm), INTENT(IN) :: ER, INTENT(IN) :: ndims AL, INTENT(IN) :: period ER, INTENT(OUT) :: newra ER, OPTIONAL, INTENT(OUT)</pre>	, dims(ndims) ds(ndims) ank	16 17 18 19 20 21 22
INTEG	MAP(COMM, NDIMS, DIMS, PH ER COMM, NDIMS, DIMS(*), AL PERIODS(*)		23 24 25
ical machin		imal" placement for the calling process on the phys- of this function is to always return the rank of the ny reordering.	26 27 28 29
riods, MPI_ MPI_ MPI_U	reorder, comm_cart), with re CART_MAP(comm, ndims, COMM_SPLIT(comm, color,	key, comm_cart), with color = 0 if newrank $\neq$ DEFINED otherwise, and key = newrank. If ndims	30 31 32 33 34 35 36
The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.			30 37 38 39 40
		ons can be implemented locally, using the topology e communicator. ( <i>End of advice to implementors.</i> )	41 42
The co	prresponding function for grap	bh structures is as follows.	43 44 45

MPI_CART_MAP(c	mm. ndims.	dims.	periods.	newrank	۱
----------------	------------	-------	----------	---------	---

1 MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank) 2 IN comm input communicator (handle) 3 IN number of graph nodes (integer) nnodes 4 5IN index integer array specifying the graph structure, see 6 MPI\_GRAPH\_CREATE 7 IN edges integer array specifying the graph structure 8 OUT newrank reordered rank of the calling process; 9 MPI\_UNDEFINED if the calling process does not be-10 long to graph (integer) 11 12int MPI\_Graph\_map(MPI\_Comm comm, int nnodes, const int index[], const 13 int edges[], int \*newrank) 1415MPI\_Graph\_map(comm, nnodes, index, edges, newrank, ierror) BIND(C) 16TYPE(MPI\_Comm), INTENT(IN) :: comm 17INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(\*) 18 INTEGER, INTENT(OUT) :: newrank 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_GRAPH\_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 21INTEGER COMM, NNODES, INDEX(\*), EDGES(\*), NEWRANK, IERROR 22 23Advice to implementors. The function MPI\_GRAPH\_CREATE(comm, nnodes, index,  $^{24}$ edges, reorder, comm\_graph), with reorder = true can be implemented by calling 2526MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank), then calling MPI\_COMM\_SPLIT(comm, color, key, comm\_graph), with color = 0 if newrank  $\neq$ 27MPI\_UNDEFINED, color = MPI\_UNDEFINED otherwise, and key = newrank. 2829 All other graph topology functions can be implemented locally, using the topology 30 information that is cached with the communicator. (End of advice to implementors.)  $^{31}$ 32 33 7.6 Neighborhood Collective Communication on Process Topologies 3435

<sup>35</sup> MPI process topologies specify a communication graph, but they implement no commu-<sup>36</sup> nication function themselves. Many applications require sparse nearest neighbor commu-<sup>37</sup> nications that can be expressed as graph topologies. We now describe several collective <sup>38</sup> operations that perform communication along the edges of a process topology. All of these <sup>39</sup> functions are collective; i.e., they must be called by all processes in the specified com-<sup>40</sup> municator. See Section 5 on page 141 for an overview of other dense (global) collective <sup>41</sup> communication operations and the semantics of collective operations.

<sup>42</sup> If the graph was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT with sources <sup>43</sup> and destinations containing 0, ..., n-1, where n is the number of processes in the group <sup>44</sup> of comm\_old (i.e., the graph is fully connected and also includes an edge from each node <sup>45</sup> to itself), then the sparse neighborhood communication routine performs the same data <sup>46</sup> exchange as the corresponding dense (fully-connected) collective operation. In the case of a <sup>47</sup> Cartesian communicator, only nearest neighbor communication is provided, corresponding <sup>48</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.*)

 $\mathbf{2}$ 

 $^{24}$ 

For a distributed graph topology, created with MPI\_DIST\_GRAPH\_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined as the sequence returned by MPI\_DIST\_GRAPH\_NEIGHBORS for destinations and sources, respectively. For a general graph topology, created with MPI\_GRAPH\_CREATE, the order of neighbors in the send and receive buffers is defined as the sequence of neighbors as returned by MPI\_GRAPH\_NEIGHBORS. Note that general graph topologies should generally be replaced by the distributed graph topologies.

For a Cartesian topology, created with MPI\_CART\_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined by order of the dimensions, first the neighbor in the negative direction and then in the positive direction with displacement 1. The numbers of sources and destinations in the communication routines are 2\*ndims with ndims defined in MPI\_CART\_CREATE. If a neighbor does not exist, i.e., at the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., periods[...]==false), then this neighbor is defined to be MPI\_PROC\_NULL.

If a neighbor in any of the functions is MPI\_PROC\_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI\_PROC\_NULL in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

# 7.6.1 Neighborhood Gather

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.

```
1
     MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
\mathbf{2}
                    comm)
3
       IN
                 sendbuf
                                             starting address of send buffer (choice)
4
       IN
                 sendcount
                                             number of elements sent to each neighbor (non-negative
5
                                             integer)
6
7
       IN
                 sendtype
                                             data type of send buffer elements (handle)
8
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
9
       IN
                                             number of elements received from each neighbor (non-
                 recvcount
10
                                             negative integer)
11
12
       IN
                 recvtype
                                             data type of receive buffer elements (handle)
13
                                             communicator with topology structure (handle)
       IN
                 comm
14
15
     int MPI_Neighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype
16
                    sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
17
                    MPI_Comm comm)
18
19
     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
20
                    recvtype, comm, ierror) BIND(C)
21
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
22
          TYPE(*), DIMENSION(..) :: recvbuf
23
          INTEGER, INTENT(IN) :: sendcount, recvcount
24
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
28
                    RECVTYPE, COMM, IERROR)
29
          <type> SENDBUF(*), RECVBUF(*)
30
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
^{31}
32
          This function supports Cartesian communicators, graph communicators, and distributed
33
     graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph
34
     communicator, the outcome is as if each process executed sends to each of its outgoing
35
     neighbors and receives from each of its incoming neighbors:
36
37
     MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
38
     int *srcs=(int*)malloc(indegree*sizeof(int));
39
     int *dsts=(int*)malloc(outdegree*sizeof(int));
40
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
41
                                  outdegree,dsts,MPI_UNWEIGHTED);
42
     int k,l;
43
44
     /* assume sendbuf and recvbuf are of type (char*) */
45
     for(k=0; k<outdegree; ++k)</pre>
46
       MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
47
48
     for(l=0; l<indegree; ++1)</pre>
```

 $\overline{7}$ 

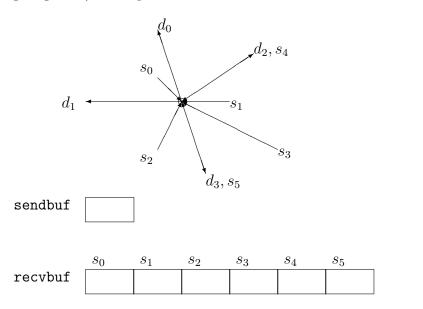
 $^{24}$ 

 $^{31}$ 

#### 

#### MPI\_Waitall(...);

Figure 7.6.1 shows the neighborhood gather communication of one process with outgoing neighbors  $d_0 \ldots d_3$  and incoming neighbors  $s_0 \ldots s_5$ . The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.



All arguments are significant on all processes and the argument comm must have identical values on all processes.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

*Rationale.* For optimization reasons, the same type signature is required independently of whether the topology graph is connected or not. (*End of rationale.*)

The "in place" option is not meaningful for this operation.

The vector variant of MPI\_NEIGHBOR\_ALLGATHER allows one to gather different numbers of elements from each neighbor.

1 2	MPI_NEIG	HBOR_ALLGATHERV(sendbut recvtype, comm)	f, sendcount, sendtype, recvbuf, recvcounts, displs,
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)
5	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
7	IN	sendtype	data type of send buffer elements (handle)
8 9	OUT	recvbuf	starting address of receive buffer (choice)
10 11 12	IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor
13 14 15	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
16 17	IN	recvtype	data type of receive buffer elements (handle)
18	IN	comm	communicator with topology structure (handle)
19			
20 21 22 23	int MPI_N	MPI_Datatype sendtyp	<pre>void* sendbuf, int sendcount, e, void* recvbuf, const int recvcounts[], MPI_Datatype recvtype, MPI_Comm comm)</pre>
24 25 26 27 28 29 30 31	TYPE( TYPE( INTEG TYPE( TYPE(	<pre>displs, recvtype, co *), DIMENSION(), INTEN *), DIMENSION() :: re</pre>	<pre>I(IN) :: sendbuf ecvbuf ount, recvcounts(*), displs(*) 0 :: sendtype, recvtype     comm</pre>
32 33 34 35 36	<type< th=""><th>DISPLS, RECVTYPE, CO &gt;&gt; SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, H</th><th></th></type<>	DISPLS, RECVTYPE, CO >> SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, H	
37 38 39 40 41	graph com communica	municators as described in Sec	nmunicators, graph communicators, and distributed tion 7.6 on page 314. If <b>comm</b> is a distributed graph ch process executed sends to each of its outgoing ncoming neighbors:
42 43 44 45 46 47 48	int *srcs int *dsts	=(int*)malloc(indegree*s: =(int*)malloc(outdegree*s graph_neighbors(comm,inde	

```
1
                                                                                                \mathbf{2}
/* assume sendbuf and recvbuf are of type (char*) */
                                                                                                3
for(k=0; k<outdegree; ++k)</pre>
  MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
                                                                                                4
                                                                                                5
                                                                                                6
for(l=0; l<indegree; ++1)</pre>
  MPI_Irecv(recvbuf+displs[1]*extent(recvtype),recvcounts[1],recvtype,
                                                                                                7
              srcs[1],...);
                                                                                                9
                                                                                               10
MPI_Waitall(...);
                                                                                               11
    The type signature associated with sendcount, sendtype, at process j must be equal
                                                                                               12
to the type signature associated with recvcounts[1], recvtype at any other process with
                                                                                               13
srcs[1]==j. This implies that the amount of data sent must be equal to the amount of
                                                                                               14
data received, pairwise between every pair of communicating processes. Distinct type maps
                                                                                               15
between sender and receiver are still allowed. The data received from the 1-th neighbor is
                                                                                               16
placed into recvbuf beginning at offset displs[1] elements (in terms of the recvtype).
                                                                                               17
    The "in place" option is not meaningful for this operation.
                                                                                               18
    All arguments are significant on all processes and the argument
                                                                                               19
comm must have identical values on all processes.
                                                                                               20
                                                                                               21
7.6.2
       Neighbor Alltoall
                                                                                               22
                                                                                               23
In this function, each process i receives data items from each process j if an edge (j,i)
                                                                                               24
exists in the topology graph or Cartesian topology. Similarly, each process i sends data
                                                                                               25
items to all processes j where an edge (i, j) exists. This call is more general than
                                                                                               26
MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor.
                                                                                               27
The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in
                                                                                               28
the receive buffer is received from the l-th neighbor.
                                                                                               29
                                                                                               30
                                                                                               31
MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
                                                                                               32
                                                                                               33
  IN
            sendbuf
                                          starting address of send buffer (choice)
                                                                                               34
  IN
            sendcount
                                          number of elements sent to each neighbor (non-negative
                                                                                               35
                                          integer)
                                                                                               36
                                                                                               37
  IN
            sendtype
                                          data type of send buffer elements (handle)
                                                                                               38
  OUT
            recvbuf
                                          starting address of receive buffer (choice)
                                                                                               39
  IN
            recycount
                                          number of elements received from each neighbor (non-
                                                                                               40
                                                                                               41
                                          negative integer)
                                                                                               42
  IN
                                          data type of receive buffer elements (handle)
            recvtype
                                                                                               43
  IN
            comm
                                          communicator with topology structure (handle)
                                                                                               44
                                                                                               45
int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype
                                                                                               46
                sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
                                                                                               47
                MPI_Comm comm)
                                                                                               48
```

```
1
     MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
\mathbf{2}
                    recvtype, comm, ierror) BIND(C)
3
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
4
          TYPE(*), DIMENSION(..) :: recvbuf
5
          INTEGER, INTENT(IN) :: sendcount, recvcount
6
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
          TYPE(MPI_Comm), INTENT(IN) :: comm
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
10
                    RECVTYPE, COMM, IERROR)
11
          <type> SENDBUF(*), RECVBUF(*)
12
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
13
14
         This function supports Cartesian communicators, graph communicators, and distributed
15
     graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph
16
     communicator, the outcome is as if each process executed sends to each of its outgoing
17
     neighbors and receives from each of its incoming neighbors:
18
19
     MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
     int *srcs=(int*)malloc(indegree*sizeof(int));
20
21
     int *dsts=(int*)malloc(outdegree*sizeof(int));
22
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
23
                                 outdegree,dsts,MPI_UNWEIGHTED);
^{24}
     int k,l;
25
26
     /* assume sendbuf and recvbuf are of type (char*) */
27
     for(k=0; k<outdegree; ++k)</pre>
       MPI_Isend(sendbuf+k*sendcount*extent(sendtype),sendcount,sendtype,
28
29
                   dsts[k],...);
30
^{31}
     for(1=0; 1<indegree; ++1)</pre>
32
       MPI_Irecv(recvbuf+l*recvcount*extent(recvtype),recvcount,recvtype,
33
                   srcs[1],...);
34
35
     MPI_Waitall(...);
36
         The type signature associated with sendcount, sendtype, at a process must be equal to
37
     the type signature associated with recvcount, recvtype at any other process. This implies
38
     that the amount of data sent must be equal to the amount of data received, pairwise between
39
     every pair of communicating processes. Distinct type maps between sender and receiver are
40
     still allowed.
41
         The "in place" option is not meaningful for this operation.
42
         All arguments are significant on all processes and the argument
43
     comm must have identical values on all processes.
44
         The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
45
     numbers of elements to and from each neighbor.
46
47
48
```

MPI_NEIG	HBOR_ALLTOALLV(sendbuf, s rdispls, recvtype, comm)	sendcounts, sdispls, sendtype, recvbuf, recvcounts,	1 $2$
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 to send the outgoing data to neighbor j	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	starting address of receive buffer (choice)	11 12
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from	13 14
		each neighbor	15
IN	rdispls	integer array (of length indegree). Entry i specifies	16
		the displacement (relative to $recvbuf)$ at which to place	17 18
		the incoming data from neighbor <b>i</b>	19
IN	recvtype	data type of receive buffer elements (handle)	20
IN	comm	communicator with topology structure (handle)	21
			22 23
int MPI_N	•	<pre>roid* sendbuf, const int sendcounts[],</pre>	24
	-	<pre>MPI_Datatype sendtype, void* recvbuf, [], const int rdispls[], MPI_Datatype</pre>	25
	recvtype, MPI_Comm co		26
	• -		27
MP1_Neign		ndcounts, sdispls, sendtype, recvbuf, recvtype, comm, ierror) BIND(C)	28 29
TYPE(	*), DIMENSION(), INTENT		29 30
		cvbuf	31
INTEG	ER, INTENT(IN) :: sendco	<pre>unts(*), sdispls(*), recvcounts(*),</pre>	32
rdisp			33
	MPI_Datatype), INTENT(IN)		34
	MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)	comm :: ierror	35
INIEG	ER, OFIIONAL, INIENI(001)	101101	$\frac{36}{37}$
MPI_NEIGH		NDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	38
~ 1		RECVTYPE, COMM, IERROR)	39
01	<pre>&gt; SENDBUF(*), RECVBUF(*) EP SENDCOUNTS(*) SDISDIS</pre>	(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	40
	YPE, COMM, IERROR	(*), DEMUTITE, RECVCUUMID $(*)$ , RUIDED $(*)$ ,	41
			42
		amunicators, graph communicators, and distributed cion 7.6 on page 314. If comm is a distributed graph	43 44

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:

45

 $46 \\ 47$ 

48

MPI\_Dist\_graph\_neighbors\_count(comm,&indegree,&outdegree,&weighted);

```
1
      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);
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+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
10
                    dsts[k],...);
11
12
     for(l=0; l<indegree; ++1)</pre>
        MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype),recvcounts[1],recvtype,
13
14
                    srcs[1],...);
15
16
     MPI_Waitall(...);
17
          The type signature associated with sendcounts [k], sendtype with dsts [k] == j at pro-
18
      cess i must be equal to the type signature associated with recvcounts[1], recvtype with
19
      srcs[1]==i at process j. This implies that the amount of data sent must be equal to the
20
21
      amount of data received, pairwise between every pair of communicating processes. Distinct
      type maps between sender and receiver are still allowed. The data in the sendbuf beginning
22
      at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor.
23
      The data received from the 1-th incoming neighbor is placed into recvbuf beginning at offset
^{24}
25
      rdispls[1] elements (in terms of the recvtype).
26
          The "in place" option is not meaningful for this operation.
          All arguments are significant on all processes and the argument
27
      comm must have identical values on all processes.
28
          MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes
29
      to and from each neighbor.
30
^{31}
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_NE	IGHBOR_ALLTOALLW(s rdispls, recvtypes	sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, <sup>1</sup> , comm) <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	
IN	sdispls	integer array (of length outdegree). Entry j specifies 7 the displacement in bytes (relative to sendbuf) from 8 which to take the outgoing data destined for neighbor 9 j (array of integers) 10	)
IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)	2
OUT	recvbuf	starting address of receive buffer (choice) 15	
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	7
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)	) L 2
IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles) 23	1 5
IN	comm	communicator with topology structure (handle) 27 28	
int MPI	const MPI_Aint void* recvbuf	const void* sendbuf, const int sendcounts[],29c sdispls[], const MPI_Datatype sendtypes[],30, const int recvcounts[], const MPI_Aint31nst MPI_Datatype recvtypes[], MPI_Comm comm)32	€)    2
TYP TYP INT INT TYP TYP	recvcounts, ro E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) :: EGER(KIND=MPI_ADDRES	sendcounts(*), recvcounts(*)38SS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)39ENT(IN) :: sendtypes(*), recvtypes(*)40IN) :: comm41	1 5 7 3 9
<ty INT INT</ty 	RECVCOUNTS, RI rpe> SENDBUF(*), REC EGER(KIND=MPI_ADDRES	BUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,43DISPLS, RECVTYPES, COMM, IERROR)44	3 1 5 7

<sup>1</sup> This function supports Cartesian communicators, graph communicators, and distributed <sup>2</sup> graph communicators as described in Section 7.6 on page 314. If comm is a distributed graph <sup>3</sup> communicator, the outcome is as if each process executed sends to each of its outgoing <sup>4</sup> neighbors and receives from each of its incoming neighbors:

```
6
     MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
     int *srcs=(int*)malloc(indegree*sizeof(int));
7
     int *dsts=(int*)malloc(outdegree*sizeof(int));
8
9
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
10
                                outdegree,dsts,MPI_UNWEIGHTED);
11
     int k,l;
12
     /* assume sendbuf and recvbuf are of type (char*) */
13
     for(k=0; k<outdegree; ++k)</pre>
14
       MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);
15
16
17
     for(l=0; l<indegree; ++1)</pre>
       MPI_Irecv(recvbuf+rdispls[1], recvcounts[1], recvtypes[1], srcs[1], ...);
18
19
     MPI_Waitall(...);
20
```

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.

27 28

21

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.

29 30 31

32 33

34

35

# 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.

- 45 46
- 47
- 48

7.7.1	Nonblocking Neighborhood Ga	ther	1
			2 3
MPI_II	NEIGHBOR_ALLGATHER(sendbu comm, request)	f, sendcount, sendtype, recvbuf, recvcount, recvtype,	4 5
IN	sendbuf	starting address of send buffer (choice)	6 7
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	starting address of receive buffer (choice)	11 12
IN	recvcount	number of elements received from each neighbor (non-negative integer)	13 14
IN	recvtype	data type of receive buffer elements (handle)	15
IN	comm	communicator with topology structure (handle)	16 17
OUT	request	communication request (handle)	18
די די וח די די		CHRONOUS :: recvbuf count, recvcount N) :: sendtype, recvtype : comm T) :: request	23 24 25 26 27 28 29 30 31
MPI_IN < IN	NEIGHBOR_ALLGATHER(SENDBUF, RECVTYPE, COMM, REQ type> SENDBUF(*), RECVBUF( NTEGER SENDCOUNT, SENDTYPE,	SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, UEST, IERROR)	32 33 34 35 36 37 38 39 40 41 42 43 44
			45 46

sendbuf sendcount sendtype recvbuf recvcounts displs	<ul> <li>starting address of send buffer (choice)</li> <li>number of elements sent to each neighbor (non-negative integer)</li> <li>data type of send buffer elements (handle)</li> <li>starting address of receive buffer (choice)</li> <li>non-negative integer array (of length indegree) containing the number of elements that are received from</li> </ul>
sendtype recvbuf recvcounts	<ul> <li>integer)</li> <li>data type of send buffer elements (handle)</li> <li>starting address of receive buffer (choice)</li> <li>non-negative integer array (of length indegree) containing the number of elements that are received from</li> </ul>
recvbuf recvcounts	starting address of receive buffer (choice) non-negative integer array (of length indegree) con- taining the number of elements that are received from
recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from
	taining the number of elements that are received from
	taining the number of elements that are received from
displa	each neighbor
աթթ	integer array (of length indegree). Entry <b>i</b> specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from neighbor <b>i</b>
recvtype	data type of receive buffer elements (handle)
comm	communicator with topology structure (handle)
request	communication request (handle)
9     OUT request     communication request (handle)       91     int MPI_Ineighbor_allgatherv(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)       96     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) BIND(C)       97     TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf       98     TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf       99     INTEGER, INTENT(IN) :: sendcount       90     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)       91     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype       93     TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype       94     TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype       95     TYPE(MPI_Request), INTENT(OUT) :: ierror       96     MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVEOUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR)       97     SENDBUF(*), RECVBUF(*)       98     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR       99     This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV.	
	<pre>comm request _Ineighbor_allgatherv(com     MPI_Datatype sendty     const int displs[]     MPI_Request *reques ighbor_allgatherv(sendbuf     displs, recvtype, o E(*), DIMENSION(), INTE E(*), DIMENSION(), ASYN EGER, INTENT(IN) :: send EGER, INTENT(IN), ASYNCHR E(MPI_Datatype), INTENT(I E(MPI_Comm), INTENT(IN) :: E(MPI_Request), INTENT(OU EGER, OPTIONAL, INTENT(OU IGHBOR_ALLGATHERV(SENDBUF     DISPLS, RECVTYPE, ( 'pe&gt; SENDBUF(*), RECVBUF() EGER SENDCOUNT, SENDTYPE, UEST, IERROR</pre>

7.7.2 No	onblocking Neighborh	ood Alltoall	1
			2 3
MPI_INEI	GHBOR_ALLTOALL(so request)	endbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,	5
IN	sendbuf	starting address of send buffer (choice)	6 7
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	starting address of receive buffer (choice)	11 12
IN	recvcount	number of elements received from each neighbor (non-negative integer)	13 14
IN	recvtype	data type of receive buffer elements (handle)	15
IN	comm	communicator with topology structure (handle)	16 17
OUT	request	communication request (handle)	18
TYPE TYPE INTE TYPE TYPE TYPE	MPI_Comm comm ghbor_alltoall(send recvtype, com (*), DIMENSION() (*), DIMENSION() GER, INTENT(IN) :: (MPI_Datatype), INT (MPI_Comm), INTENT	sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: request	22 23 24 25 26 27 28 29 30 31 32
<typ INTE</typ 	RECVTYPE, COM De> SENDBUF(*), REG GER SENDCOUNT, SENI	DBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, M, REQUEST, IERROR) CVBUF(*) DTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR ng variant of MPI_NEIGHBOR_ALLTOALL.	<ul> <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> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> </ul>

<pre>N sendbuf starting address of send buffer (choice) N sendcounts non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor N sdispls integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j N sendtype data type of send buffer elements (handle) OUT recvluf starting address of receive buffer (choice) N recvcounts non-negative integer array (of length indegree) speci- fying the number of elements that are received from cach neighbor N recvcunts integer array (of length indegree). Entry i specifies the displacement (relative to sendbuff i specifies the displacement (relative to recowf) at which to place the incoming data from neighbor i N recvtype data type of receive buffer elements (handle) N comm communicator with topology structure (handle) OUT request communication request (handle) int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>	12	MPI_INEI	GHBOR_ALLTOALLV(sendbuf, rdispls, recvtype, comm,	sendcounts, sdispls, sendtype, recvbuf, recvcounts, request)
IN       sendcounts       non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor         IN       sdispls       integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j         IN       sendtype       data type of send buffer elements (handle)         OUT       recvounts       non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor         IN       recvcounts       non-negative integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i         IN       recvtype       data type of receive buffer elements (handle)         IN       recvtype       data type of receive buffer elements (handle)         IN       recvtype       data type of receive buffer elements (handle)         IN       recvtype       data type of receive buffer elements (handle)         IN       comm       communicator with topology structure (handle)         OUT       request       communication request (handle)         IN       comm       communication request (handle)         IN       comm       communication request (handle)         OUT       request       communication request (handle)         IN       const int recvoo		IN	sendbuf	starting address of send buffer (choice)
<pre>Int subply integer and (to teger Undegree). That y specifies the displacement (relative to sendbuff from which send the outgoing data to neighbor j IN sendtype data type of send buffer elements (handle) OUT recvbuf starting address of receive buffer (choice) IN recvcounts non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor IN rdispls integer array (of length indegree). Entry i specifies the displacement (relative to recvbuff at which to place the incoming data from neighbor i IN recvtype data type of receive buffer elements (handle) OUT request communicator with topology structure (handle) OUT request communication request (handle) IN Comm communication request (handle) IN Comm communication request (handle) IN Comm communication request *request) MPI_Ineighbor_alltoallv(const void* sendbuff, const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) MPI_Ineighbor), INTENT(IN), ASYNCHRONOUS :: sendbuff TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuff TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuff TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(IN) :: request INTEGER SENDEQUENTS, RECVTYPE, COMM, REQUEST, IERROR </pre>	5	IN	sendcounts	
IN       sendtype       data type of send buffer elements (handle)         12       OUT       recvbuf       starting address of receive buffer (choice)         13       IN       recvcounts       non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor         14       IN       rdspls       integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i         16       IN       recvtype       data type of receive buffer elements (handle)         17       IN       recvtype       data type of receive buffer elements (handle)         18       IN       recvtype       data type of receive buffer elements (handle)         19       IN       recvtype       data type of receive buffer elements (handle)         10       COUT       request       communicator with topology structure (handle)         20       OUT       request       communication request (handle)         21       Int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[], const int stispls[], MPI_Datatype sendtype, void* recvbuf, recvtype, MPI_Coum comm, MPI_Request *:request)         22       IntEmSION(), ASYNCHRONOUS :: sendtype, recvbuf, recvtype, NPI_Coum comm, MPI_Request *:request)         23       INTEMER, INTENT(IN), ASYNCHRONOUS :: sendtype, recvtype <t< td=""><th>8 9</th><td>IN</td><td>sdispls</td><td>the displacement (relative to <math display="inline">{\sf sendbuf})</math> from which send</td></t<>	8 9	IN	sdispls	the displacement (relative to ${\sf sendbuf})$ from which send
12       OUT recvbuf       starting address of receive buffer (choice)         13       IN recvcounts       non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor         14       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         15       IN recvtype       data type of receive buffer elements (handle)         16       IN comm       communicator with topology structure (handle)         17       OUT request       communicator int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)         18       MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)         17       TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf         18       INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)         17       TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype         18       INTEGER, OPTIONAL, INTENT(IN) :: sendtype, recvtype         19       TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype         19       TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype         19       TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype         10       TYPE(MPI_Datat		IN	sendtype	data type of send buffer elements (handle)
<pre>fying the number of elements that are received from each neighbor integer array (of length indegree). Entry i specifies the displacement (relative to recobuf) at which to place the incoming data from neighbor i N recvtype data type of receive buffer elements (handle) N comm communication request (handle) OUT request communication request (handle) int MPI_Ineighbor_alltoallv(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) MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: ierror MPI_INEIGER, ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.</type></pre>		OUT	recvbuf	starting address of receive buffer (choice)
IN       rdispls       integer array (of length indegree). Entry i specifies         integer array (of length indegree). Entry i specifies       the displacement (relative to recvbuf) at which to place         int       recvtype       data type of receive buffer elements (handle)         int       comm       communicator with topology structure (handle)         int       MPI_Ineighbor_alltoallv(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)       recvtouf,         mecvtype(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf         TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendcounts(*), sdispls(*),         recvcounts(*), rdispls(*)         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype         TYPE(MPI_Datatype), INTENT(IN) :: comm         TYPE(MPI_Request), INTENT(OUT) :: request         INTEGER, OPTIONAL, INTENT(OUT) :: ierror         MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,         RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),         RECVTYPE, COMM, REQUEST, IERROR         This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.    <th>14 15</th><td>IN</td><td>recvcounts</td><td>fying the number of elements that are received from</td></type>	14 15	IN	recvcounts	fying the number of elements that are received from
IN       recvtype       data type of receive buffer elements (handle)         IN       comm       communicator with topology structure (handle)         OUT       request       communication request (handle)         int       MPI_Ineighbor_alltoallv(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)         MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)         TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf         TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)         TYPE(MPI_Datatype), INTENT(IN) :: comm         TYPE(MPI_Request), INTENT(IN) :: sendtype, recvtype         TYPE(MPI_Request), INTENT(IN) :: request         INTEGER, OPTIONAL, INTENT(OUT) :: ierror         MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVEUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR <type> SENDBUF(*), RECVBUF(*)         MPI_INEIGHBOR_ALLTOALLV(SENDBUF, IERROR         This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.</type>	17 18	IN	rdispls	the displacement (relative to $recvbuf)$ at which to place
OUT       request       communication request (handle)         int MPI_Ineighbor_alltoallv(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)         MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)         TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf         TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Request), INTENT(IN) :: sendtype, recvtype         TYPE(MPI_Bequest), INTENT(IN) :: sendtype, recvtype         TYPE(MPI_Request), INTENT(IN) :: ierror         MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVEUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVEUF(*)         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR         This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.</type>		IN	recvtype	data type of receive buffer elements (handle)
OUT         request         communication request (handle)           int MPI_Ineighbor_alltoallv(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)           MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)           TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf           INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)           TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: sendtype, recvtype           TYPE(MPI_Comm), INTENT(IN) :: comm           TYPE(MPI_Request), INTENT(OUT) :: ierror           MPI_INEIGHBOR_ALLTOALLV(SENDEUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVEUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)           Vipe> SENDEUF(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR           INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR	21	IN	comm	communicator with topology structure (handle)
<pre>int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],</pre>		OUT	request	communication request (handle)
<ul> <li>MFI_INEIGNOUL_AITCOALIV(SENDUAL, SENDCOUNTS, STADIS, SENDTYPE, FECUDI, recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)</li> <li>TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf</li> <li>TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf</li> <li>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),</li> <li>recvcounts(*), rdispls(*)</li> <li>TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype</li> <li>TYPE(MPI_Comm), INTENT(IN) :: comm</li> <li>TYPE(MPI_Request), INTENT(OUT) :: request</li> <li>INTEGER, OPTIONAL, INTENT(OUT) :: ierror</li> <li>MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)</li> <li><type> SENDBUF(*), RECVBUF(*)</type></li> <li>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</li> <li>This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.</li> </ul>	26 27	int MPI_I	<pre>const int sdispls[], const int recvcounts</pre>	<pre>MPI_Datatype sendtype, void* recvbuf, [], const int rdispls[], MPI_Datatype</pre>
	30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	TYPE( TYPE( INTEG recvo TYPE( TYPE( INTEG MPI_INEIG RECVT	recvcounts, rdispls, (*), DIMENSION(), INTEN (*), DIMENSION(), ASYNCH ER, INTENT(IN), ASYNCHRON counts(*), rdispls(*) MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) HBOR_ALLTOALLV(SENDBUF, S RECVCOUNTS, RDISPLS, (*) SENDBUF(*), RECVBUF(*) ER SENDCOUNTS(*), SDISPLS YPE, COMM, REQUEST, IERRO	<pre>recvtype, comm, request, ierror) BIND(C) T(IN), ASYNCHRONOUS :: sendbuf HRONOUS :: recvbuf NOUS :: sendcounts(*), sdispls(*), ) :: sendtype, recvtype comm ) :: request ) :: ierror SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVTYPE, COMM, REQUEST, IERROR) ) S(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), OR</pre>
	47 48			

	sendbuf	starting address of sond buffer (shoise)
N		starting address of send buffer (choice)
N	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
N	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)
N	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)
JUT	recvbuf	starting address of receive buffer (choice)
N	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
N	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)
N	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)
N	comm	communicator with topology structure (handle)
JUT	request	communication request (handle)
	Ineighbor_alltoallw_ const MPI_Aint void* recvbuf,	<pre>(const void* sendbuf, const int sendcounts[], t sdispls[], const MPI_Datatype sendtypes[], , const int recvcounts[], const MPI_Aint hst MPI_Datatype recvtypes[], MPI_Comm comm,</pre>
t MPI_ I_Inei TYPF	Ineighbor_alltoallw const MPI_Aint void* recvbuf, rdispls[], con MPI_Request *r ighbor_alltoallw(sen recvcounts, rd E(*), DIMENSION(),	<pre>c(const void* sendbuf, const int sendcounts[], t sdispls[], const MPI_Datatype sendtypes[], , const int recvcounts[], const MPI_Aint nst MPI_Datatype recvtypes[], MPI_Comm comm, request) dbuf, sendcounts, sdispls, sendtypes, recvbuf, displs, recvtypes, comm, request, ierror) BIND(C) INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>
I_Inei TYPI TYPI INTI INTI sdis	_Ineighbor_alltoallw const MPI_Aint void* recvbuf, rdispls[], con MPI_Request *r ighbor_alltoallw(sen recvcounts, rd E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN), AS EGER(KIND=MPI_ADDRES spls(*), rdispls(*)	<pre>(const void* sendbuf, const int sendcounts[], t sdispls[], const MPI_Datatype sendtypes[], , const int recvcounts[], const MPI_Aint hst MPI_Datatype recvtypes[], MPI_Comm comm, request) dbuf, sendcounts, sdispls, sendtypes, recvbuf, displs, recvtypes, comm, request, ierror) BIND(C) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf YNCHRONOUS :: sendcounts(*), recvcounts(*) S_KIND), INTENT(IN), ASYNCHRONOUS ::</pre>
t MPI_ I_Inei TYPE INTE INTE Sdis TYPE	_Ineighbor_alltoallw const MPI_Aint void* recvbuf, rdispls[], con MPI_Request *r ighbor_alltoallw(sen recvcounts, rd E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN), AS EGER(KIND=MPI_ADDRES spls(*), rdispls(*)	<pre>c(const void* sendbuf, const int sendcounts[], c sdispls[], const MPI_Datatype sendtypes[], c const int recvcounts[], const MPI_Aint nst MPI_Datatype recvtypes[], MPI_Comm comm, cequest) dbuf, sendcounts, sdispls, sendtypes, recvbuf, displs, recvtypes, comm, request, ierror) BIND(C) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf YNCHRONOUS :: sendcounts(*), recvcounts(*)</pre>
t MPI_ I_Inei TYPH INTH Sdis TYPH recv TYPH	_Ineighbor_alltoallw const MPI_Aint void* recvbuf, rdispls[], con MPI_Request *r ighbor_alltoallw(sen recvcounts, rd E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN), AS EGER(KIND=MPI_ADDRES spls(*), rdispls(*) E(MPI_Datatype), INT	<pre>(const void* sendbuf, const int sendcounts[], t sdispls[], const MPI_Datatype sendtypes[], , const int recvcounts[], const MPI_Aint hst MPI_Datatype recvtypes[], MPI_Comm comm, request) dbuf, sendcounts, sdispls, sendtypes, recvbuf, displs, recvtypes, comm, request, ierror) BIND(C) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf YNCHRONOUS :: sendcounts(*), recvcounts(*) S_KIND), INTENT(IN), ASYNCHRONOUS :: ENT(IN), ASYNCHRONOUS :: sendtypes(*), IN) :: comm</pre>

1 2 3 4 5 6 7 8 9	<pre>MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>
10 11	7.8 An Application Example
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	<b>Example 7.9</b> The example in Figures 7.1-7.3 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)).
30	
31 32	
33	
34	
35	
36	
37	
38 39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

```
6
                                                                                     7
INTEGER ndims, num_neigh
                                                                                     8
LOGICAL reorder
                                                                                     9
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                    10
                                                                                    11
INTEGER comm, comm_cart, dims(ndims), ierr
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
                                                                                    12
LOGICAL periods(ndims)
                                                                                    13
REAL u(0:101,0:101), f(0:101,0:101)
                                                                                    14
DATA dims / ndims * 0 /
                                                                                    15
comm = MPI_COMM_WORLD
                                                                                    16
                                                                                    17
!
    Set process grid size and periodicity
                                                                                    18
CALL MPI_DIMS_CREATE(comm, ndims, dims,ierr)
periods(1) = .TRUE.
                                                                                    19
periods(2) = .TRUE.
                                                                                    20
    Create a grid structure in WORLD group and inquire about own position
                                                                                    21
CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
                                                                                    22
                   comm_cart,ierr)
                                                                                    23
                                                                                    ^{24}
CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords,ierr)
                                                                                    25
i = own_coords(1)
                                                                                    26
j = own_coords(2)
    Look up the ranks for the neighbors. Own process coordinates are (i,j).
                                                                                    27
Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
                                                                                    28
CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
                                                                                    29
CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3),neigh_rank(4), ierr)
                                                                                    30
    Initialize the grid functions and start the iteration
                                                                                    31
!
CALL init (u, f)
                                                                                    32
                                                                                    33
DO it=1,100
                                                                                    34
   CALL relax (u, f)
!
       Exchange data with neighbor processes
                                                                                    35
   CALL exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                    36
END DO
                                                                                    37
CALL output (u)
                                                                                    38
                                                                                    39
                                                                                    40
   Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
```

```
1
\mathbf{2}
3
4
5
6
7
8
9
10
11
     SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
12
     REAL u(0:101,0:101)
13
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
14
     REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
15
     INTEGER ierr
16
     sndbuf(1:100,1) = u( 1,1:100)
17
     sndbuf(1:100,2) = u(100,1:100)
18
     sndbuf(1:100,3) = u(1:100, 1)
19
     sndbuf(1:100,4) = u(1:100,100)
20
     CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
21
                                  comm_cart, ierr)
22
     ! instead of
23
     ! DO i=1,num_neigh
24
         CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),ierr)
     !
25
         CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),ierr)
     !
26
     ! END DO
27
     ! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
28
29
     u( 0,1:100) = rcvbuf(1:100,1)
30
     u(101,1:100) = rcvbuf(1:100,2)
31
     u(1:100, 0) = rcvbuf(1:100,3)
32
     u(1:100,101) = rcvbuf(1:100,4)
33
     END
34
35
36
     Figure 7.2: Communication routine with local data copying and sparse neighborhood all-
37
     to-all.
38
39
40
41
42
43
44
45
46
47
48
```

```
2
                                                                                  3
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
USE MPI
                                                                                  4
REAL u(0:101,0:101)
                                                                                  5
                                                                                  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)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal, sdispls(num_neigh), &
                                                                                  9
                                                                                  10
                                 rdispls(num_neigh)
                                                                                  11
INTEGER type_vec, i, ierr
    The following initialization need to be done only once
                                                                                  12
                                                                                  13
Т
    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
sndtypes(4) = MPI_REAL
                                                                                  20
                                                                                  21
DO i=1,num_neigh
   sndcounts(i) = 100
                                                                                  22
                                                                                  23
   rcvcounts(i) = 100
                                                                                  24
   rcvtypes(i) = sndtypes(i)
                                                                                  25
END DO
                                                                                  26
                                              ! first element of u( 1,1:100)
sdispls(1) = (1 +
                     1*102) * sizeofreal
sdispls(2) = (100 + 1*102) * sizeofreal
                                              ! first element of u(100,1:100)
                                                                                  27
sdispls(3) = (1 + 1*102) * size of real ! first element of u(1:100, 1)
                                                                                  28
                                                                            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)
rdispls(3) = (1 + 0*102) * size of real ! first element of u(1:100, )
                                                                                  32
                                                                            0)
                                                                                  33
rdispls(4) = ( 1 + 101*102) * sizeofreal ! first element of u(1:100,101)
                                                                                  34
! the following communication has to be done in each call of exchange
                                                                                  35
                                                                                  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
    after the last call of exchange.
                                                                                  40
!
                                                                                  41
CALL MPI_TYPE_FREE (type_vec, ierr)
                                                                                  42
END
                                                                                  43
                                                                                  44
Figure 7.3: Communication routine with sparse neighborhood all-to-all-w and without local
                                                                                  45
data copying.
                                                                                  46
```

# Chapter 8

# **MPI** Environmental Management

 $\frac{24}{25}$ 

 $41 \\ 42$ 

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

# 8.1 Implementation Information

### 8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C,

#define MPI\_VERSION 3
#define MPI\_SUBVERSION 0

in Fortran,

INTEGER ::	MPI_VERSION, M	IPI_SUBVERSION
PARAMETER	(MPI_VERSION	= 3)
PARAMETER	(MPI_SUBVERSION	1 = 0)

For runtime determination,

```
MPI_GET_VERSION( version, subversion )
```

```
OUTversionversion number (integer)OUTsubversionsubversion number (integer)int MPI_Get_version(int *version, int *subversion)
```

```
      MPI_Get_version(version, subversion, ierror) BIND(C)
      46

      INTEGER, INTENT(OUT) :: version, subversion
      48
```

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2 3 4	MPI_GET_VERSION(VERSION, SUBVERSION, IERROR) INTEGER VERSION, SUBVERSION, IERROR
5 6 7 8 9	MPI_GET_VERSION can be called before MPI_INIT and after MPI_FINALIZE. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (3,0), (2,2), (2,1), (2,0), and (1,2).
10 11	MPI_GET_LIBRARY_VERSION( version, resultlen )
12	OUT version version string (string)
13 14 15	OUT     resultlen     Length (in printable characters) of the result returned in version (integer)
16 17	<pre>int MPI_Get_library_version(char *version, int *resultlen)</pre>
18 19 20 21 22	<pre>MPI_Get_library_version(version, resulten, ierror) BIND(C)     CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version     INTEGER, INTENT(OUT) :: resultlen     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
22 23 24 25	MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR) CHARACTER*(*) VERSION INTEGER RESULTLEN,IERROR
26 27 28	This routine returns a string representing the version of the MPI library. The version argument is a character string for maximum flexibility.
29 30 31 32	Advice to implementors. An implementation of MPI should return a different string for every change to its source code or build that could be visible to the user. ( <i>End of advice to implementors.</i> )
33 34 35	The argument version must represent storage that is MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may write up to this many characters into version.
36 37 38 39 40 41 42 43	The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING. MPI_GET_LIBRARY_VERSION_can be called before MPI_INIT and after MPI_FINALIZE.
44 45 46 47 48	8.1.2 Environmental Inquiries A set of attributes that describe the execution environment are attached to the commu- nicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_COMM_GET_ATTR described in Section 6.7 on page 265

and in Section 11.2.1 on page 000. It is croneous to delete these attributes, nee then keys,	1 $2$
	3
MPI TAG UB Upper bound for tag value	4 5
MFI_HOST Host process rank, if such exists, MFI_FROC_NOLL, otherwise.	6
<b>MPI_IO</b> rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same	7 8 9
<b>MPI_WTIME_IS_GLOBAL</b> Boolean variable that indicates whether clocks are synchronized.	10 11
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)	12 13 14 15
Advice to users. Note that in the C binding, the value returned by these attributes is a <i>nointer</i> to an int containing the requested value (End of advice to users)	16 17 18
The required parameter values are discussed in more detail below:	19 20
Tag Values	$\frac{21}{22}$
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 for MPI_TAG_UB. The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.	23 24 25 26 27 28
	29 30
The value returned for MPI_HOST gets the rank of the HOST 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 HOST, nor does it requires that a HOST exists. The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	31 32 33 34 35 36
IO Rank	37 38
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).	39 40 41 42 43

will be returned. Otherwise, if the calling process can provide language-standard I/O, then the value MPI\_ANY\_SOURCE 43 will be returned. Otherwise, if the calling process can provide language-standard I/O, 44 then its rank will be returned. Otherwise, if some process can provide language-standard 45 I/O then the rank of one such process will be returned. The same value need not be 46 returned by all processes. If no process can provide language-standard I/O, then the value 47 MPI\_PROC\_NULL will be returned. 48

```
1
           Advice to users. Note that input is not collective, and this attribute does not indicate
\mathbf{2}
           which process can or does provide input. (End of advice to users.)
3
4
     Clock Synchronization
5
     The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in
6
     MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered
7
     synchronized if explicit effort has been taken to synchronize them. The expectation is that
8
     the variation in time, as measured by calls to MPI_WTIME, will be less then one half the
9
     round-trip time for an MPI message of length zero. If time is measured at a process just
10
     before a send and at another process just after a matching receive, the second time should
11
     be always higher than the first one.
12
          The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not
13
     synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This
14
     attribute may be associated with communicators other then MPI_COMM_WORLD.
15
          The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of
16
     MPI_COMM_WORLD.
17
18
     Inquire Processor Name
19
20
21
22
     MPI_GET_PROCESSOR_NAME( name, resultlen )
23
       OUT
                                              A unique specifier for the actual (as opposed to vir-
                 name
^{24}
                                              tual) node.
25
       OUT
                 resultlen
                                              Length (in printable characters) of the result returned
26
                                              in name
27
28
     int MPI_Get_processor_name(char *name, int *resultlen)
29
30
     MPI_Get_processor_name(name, resultlen, ierror) BIND(C)
^{31}
          CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) ::
                                                                         name
32
          INTEGER, INTENT(OUT) :: resultlen
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
36
          CHARACTER*(*) NAME
37
          INTEGER RESULTLEN, IERROR
38
          This routine returns the name of the processor on which it was called at the moment
39
     of the call. The name is a character string for maximum flexibility. From this value it
40
     must be possible to identify a specific piece of hardware; possible values include "processor
41
     9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the
42
     running homogeneous system). The argument name must represent storage that is at least
43
     MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write
```

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

 $_{45}^{44}$  up to this many characters into name.

338

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

*Rationale.* This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI\_GET\_PROCESSOR\_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI\_MAX\_PROCESSOR\_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (End of advice to users.)

# 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-	26
		ger)	27
		0 /	28
IN	info	info argument (handle)	
	into	mio algamone (nanaro)	29
OUT	baseptr	pointer to beginning of memory segment allocated	30

int MPI\_Alloc\_mem(MPI\_Aint size, MPI\_Info info, void \*baseptr)

	33
<pre>MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)</pre>	34
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	35
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size	36
TYPE(MPI_Info), INTENT(IN) :: info	37
TYPE(C_PTR), INTENT(OUT) :: baseptr	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)	40
INTEGER INFO, IERROR	41
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	42

If the Fortran compiler provides TYPE(C\_PTR), then the following interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR, but with a different linker name: 47

```
1
     INTERFACE MPI_ALLOC_MEM
\mathbf{2}
          SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
3
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
4
              INTEGER :: INFO, IERROR
5
              INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
6
              TYPE(C_PTR) :: BASEPTR
7
          END SUBROUTINE
8
     END INTERFACE
9
          The linker name base of this overloaded function is MPI_ALLOC_MEM_CPTR. The
10
11
     implied linker names are described in Section 17.1.5 on page 607.
          The info argument can be used to provide directives that control the desired location
12
     of the allocated memory. Such a directive does not affect the semantics of the call. Valid
13
     info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL
14
     is always valid.
15
16
         The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
     to indicate it failed because memory is exhausted.
17
18
19
     MPI_FREE_MEM(base)
20
       IN
21
                 base
                                             initial address of memory segment allocated by
22
                                             MPI_ALLOC_MEM (choice)
23
24
     int MPI_Free_mem(void *base)
25
     MPI_Free_mem(base, ierror) BIND(C)
26
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_FREE_MEM(BASE, IERROR)
30
          <type> BASE(*)
31
          INTEGER IERROR
32
          The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to
33
     indicate an invalid base argument.
34
35
           Rationale. The C bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar
36
           to the bindings for the malloc and free C library calls: a call to
37
           MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one
38
           less level of indirection). Both arguments are declared to be of same type
39
           void<sup>*</sup> so as to facilitate type casting. The Fortran binding is consistent with the C
40
           bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the TYPE(C_PTR)
41
           pointer or the (integer valued) address of the allocated memory. The base argument
42
           of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable
43
           stored at that location. (End of rationale.)
44
45
           Advice to implementors.
                                      If MPI_ALLOC_MEM allocates special memory, then a
46
           design similar to the design of C malloc and free functions has to be used, in order
47
           to find out the size of a memory segment, when the segment is freed. If no special
48
```

memory is used, MPI\_ALLOC\_MEM simply invokes malloc, and MPI\_FREE\_MEM invokes free.

A call to MPI\_ALLOC\_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (*End of advice to implementors.*)

**Example 8.1** Example of use of MPI\_ALLOC\_MEM, in Fortran with 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 *Craypointer*. 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

```
1
       float (* f)[100][100];
2
       /* no memory is allocated */
3
       MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
4
       /* memory allocated */
5
       . . .
6
       (*f)[5][3] = 2.71;
7
8
       MPI_Free_mem(f);
9
10
```

#### 8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementationdependent. Each such error generates an **MPI** exception.

16The above text takes precedence over any text on error handling within this document. 17Specifically, text that states that errors will be handled should be read as may be handled. 18

A user can associate error handlers to three types of objects: communicators, windows, 19and files. The specified error handling routine will be used for any MPI exception that occurs 20during a call to MPI for the respective object. MPI calls that are not related to any objects 21are considered to be attached to the communicator MPI\_COMM\_WORLD. The attachment 22of error handlers to objects is purely local: different processes may attach different error 23handlers to corresponding objects.  $^{24}$ 

Several predefined error handlers are available in MPI:

- **MPI\_ERRORS\_ARE\_FATAL** The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI\_ABORT was called by the process that invoked the handler.
- **MPI\_ERRORS\_RETURN** The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

34The error handler MPI\_ERRORS\_ARE\_FATAL is associated by default with MPI\_COMM-35 \_WORLD after initialization. Thus, if the user chooses not to control error handling, every 36 error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the 39 error handler MPI\_ERRORS\_RETURN will be used. Usually it is more convenient and more 40 efficient not to test for errors after each MPI call, and have such error handled by a non trivial MPI error handler.

42After an error is detected, the state of MPI is undefined. That is, using a user-defined 43 error handler, or MPI\_ERRORS\_RETURN, does not necessarily allow the user to continue to 44use MPI after an error is detected. The purpose of these error handlers is to allow a user to 45issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O46 buffers) before a program exits. An MPI implementation is free to allow MPI to continue 47after an error but is not required to do so. 48

11 12

13

14

15

2526

27

28

29

30

 $^{31}$ 32

33

37

38

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.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C has distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI\_XXX\_CREATE\_ERRHANDLER(function, <sup>12</sup> errhandler), where XXX is, respectively, COMM, WIN, or FILE. <sup>13</sup>

An error handler is attached to a communicator, window, or file by a call to MPI\_XXX\_SET\_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER, with matching XXX. The predefined error handlers MPI\_ERRORS\_RETURN and MPI\_ERRORS\_ARE\_FATAL can be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI\_XXX\_GET\_ERRHANDLER.

The MPI function MPI\_ERRHANDLER\_FREE can be used to free an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER.

MPI\_{COMM,WIN,FILE}\_GET\_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI\_ERRHANDLER\_FREE should be called with the error handler returned from MPI\_{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.

Advice to implementors. High-quality implementation should raise an error when an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER is attached to an object of the wrong type with a call to MPI\_YYY\_SET\_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

## 8.3.1 Error Handlers for Communicators

#### 39 MPI\_COMM\_CREATE\_ERRHANDLER(comm\_errhandler\_fn, errhandler) 40 41 IN comm\_errhandler\_fn user defined error handling procedure (function) 42OUT errhandler MPI error handler (handle) 43 44int MPI\_Comm\_create\_errhandler(MPI\_Comm\_errhandler\_function 45\*comm\_errhandler\_fn, MPI\_Errhandler \*errhandler) 4647

MPI\_Comm\_create\_errhandler(comm\_errhandler\_fn, errhandler, ierror) BIND(C)

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10

11

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28

29

30

 $^{31}$ 

32

33 34

35 36

37 38

1 PROCEDURE(MPI\_Comm\_errhandler\_function) :: comm\_errhandler\_fn 2 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI\_COMM\_CREATE\_ERRHANDLER(COMM\_ERRHANDLER\_FN, ERRHANDLER, IERROR) 5EXTERNAL COMM\_ERRHANDLER\_FN 6 INTEGER ERRHANDLER. IERROR 7 8 Creates an error handler that can be attached to communicators. 9 The user routine should be, in C, a function of type MPI\_Comm\_errhandler\_function, which 10 is defined as 11typedef void MPI\_Comm\_errhandler\_function(MPI\_Comm \*, int \*, ...); 12The first argument is the communicator in use. The second is the error code to be 13 returned by the MPI routine that raised the error. If the routine would have returned 14 MPI\_ERR\_IN\_STATUS, it is the error code returned in the status for the request that caused 15the error handler to be invoked. The remaining arguments are "stdargs" arguments whose 16 number and meaning is implementation-dependent. An implementation should clearly doc-17ument these arguments. Addresses are used so that the handler may be written in Fortran. 18 19 20With the Fortran mpi\_f08 module, the user routine comm\_errhandler\_fn should be of the 21form: 22 ABSTRACT INTERFACE 23SUBROUTINE MPI\_Comm\_errhandler\_function(comm, error\_code) BIND(C)  $^{24}$ TYPE(MPI\_Comm) :: comm 25INTEGER :: error\_code 2627With the Fortran mpi module and mpif.h, the user routine COMM\_ERRHANDLER\_FN 28should be of the form: 29 SUBROUTINE COMM\_ERRHANDLER\_FUNCTION(COMM, ERROR\_CODE) 30 INTEGER COMM, ERROR\_CODE  $^{31}$ 32 33 Rationale. The variable argument list is provided because it provides an ISO-34 standard hook for providing additional information to the error handler; without this 35 hook, ISO C prohibits additional arguments. (End of rationale.) 36 Advice to users. A newly created communicator inherits the error handler that 37 is associated with the "parent" communicator. In particular, the user can specify 38 a "global" error handler for all communicators by associating this handler with the 39 communicator MPI\_COMM\_WORLD immediately after initialization. (End of advice to 40 users.) 41 4243 MPI\_COMM\_SET\_ERRHANDLER(comm, errhandler) 4445INOUT comm communicator (handle) 46 IN errhandler new error handler for communicator (handle) 4748

### 8.3. ERROR HANDLING

1 int MPI\_Comm\_set\_errhandler(MPI\_Comm comm, MPI\_Errhandler errhandler) 2 MPI\_Comm\_set\_errhandler(comm, errhandler, ierror) BIND(C) TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 MPI\_COMM\_SET\_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR 9 Attaches a new error handler to a communicator. The error handler must be either 10 a predefined error handler, or an error handler created by a call to 11 MPI\_COMM\_CREATE\_ERRHANDLER. 1213 14MPI\_COMM\_GET\_ERRHANDLER(comm, errhandler) 15IN communicator (handle) comm 16OUT 17errhandler error handler currently associated with communicator 18 (handle) 19 20int MPI\_Comm\_get\_errhandler(MPI\_Comm comm, MPI\_Errhandler \*errhandler) 21MPI\_Comm\_get\_errhandler(comm, errhandler, ierror) BIND(C) 22 TYPE(MPI\_Comm), INTENT(IN) :: comm 23TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI\_COMM\_GET\_ERRHANDLER(COMM, ERRHANDLER, IERROR) 27INTEGER COMM. ERRHANDLER. IERROR 28 Retrieves the error handler currently associated with a communicator. 29 For example, a library function may register at its entry point the current error handler 30 for a communicator, set its own private error handler for this communicator, and restore 31before exiting the previous error handler. 32 33 8.3.2 Error Handlers for Windows 34 35 36 37 MPI\_WIN\_CREATE\_ERRHANDLER(win\_errhandler\_fn, errhandler) 38 IN win\_errhandler\_fn user defined error handling procedure (function) 39 OUT errhandler MPI error handler (handle) 40 41 42int MPI\_Win\_create\_errhandler(MPI\_Win\_errhandler\_function \*win\_errhandler\_fn, MPI\_Errhandler \*errhandler) 43 44MPI\_Win\_create\_errhandler(win\_errhandler\_fn, errhandler, ierror) BIND(C) 45PROCEDURE(MPI\_Win\_errhandler\_function) :: win\_errhandler\_fn 46TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
1
     MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
\mathbf{2}
          EXTERNAL WIN_ERRHANDLER_FN
3
          INTEGER ERRHANDLER, IERROR
4
          Creates an error handler that can be attached to a window object. The user routine
5
     should be, in C, a function of type MPI_Win_errhandler_function which is defined as
6
     typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
7
8
         The first argument is the window in use, the second is the error code to be returned.
9
     With the Fortran mpi_f08 module, the user routine win_errhandler_fn should be of the form:
10
11
     ABSTRACT INTERFACE
12
       SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)
            TYPE(MPI_Win) :: win
13
14
            INTEGER :: error_code
15
16
     With the Fortran mpi module and mpif.h, the user routine WIN_ERRHANDLER_FN should
17
     be of the form:
18
     SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
19
          INTEGER WIN, ERROR_CODE
20
21
22
23
     MPI_WIN_SET_ERRHANDLER(win, errhandler)
24
       INOUT
                 win
                                            window (handle)
25
       IN
                 errhandler
                                            new error handler for window (handle)
26
27
     int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
28
29
     MPI_Win_set_errhandler(win, errhandler, ierror) BIND(C)
30
          TYPE(MPI_Win), INTENT(IN) ::
                                          win
^{31}
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
35
          INTEGER WIN, ERRHANDLER, IERROR
36
         Attaches a new error handler to a window. The error handler must be either a pre-
37
     defined error handler, or an error handler created by a call to
38
     MPI_WIN_CREATE_ERRHANDLER.
39
40
^{41}
     MPI_WIN_GET_ERRHANDLER(win, errhandler)
42
       IN
                                            window (handle)
                 win
43
       OUT
                 errhandler
                                            error handler currently associated with window (han-
44
                                            dle)
45
46
47
     int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
48
```

	_get_errhandler(win, errh		1
	E(MPI_Win), INTENT(IN) :: E(MPI_Errhandler), INTENT		2 3
	EGER, OPTIONAL, INTENT(OU		4
			5
	_GET_ERRHANDLER(WIN, ERRH EGER WIN, ERRHANDLER, IER		6
LIN I.	EGER WIN, ERRHANDLER, IER	RUR	7
Reti	rieves the error handler curren	tly associated with a window.	8
			9 10
8.3.3 E	rror Handlers for Files		11
			12
	E_CREATE_ERRHANDLER(fil	a arrhandlar fn arrhandlar)	13
	``	,	14
IN	file_errhandler_fn	user defined error handling procedure (function)	15
OUT	errhandler	MPI error handler (handle)	16
			17 18
int MPI		PI_File_errhandler_function	19
	*file_errhandler_fi	n, MPI_Errhandler *errhandler)	20
MPI_Fil	e_create_errhandler(file_	errhandler_fn, errhandler, ierror) BIND(C)	21
		er_function) :: file_errhandler_fn	22
	E(MPI_Errhandler), INTENT		23
INT	EGER, OPTIONAL, INTENT(OU	T) :: ierror	24
MPI_FIL	E_CREATE_ERRHANDLER(FILE_	ERRHANDLER_FN, ERRHANDLER, IERROR)	25 26
EXT	ERNAL FILE_ERRHANDLER_FN		20
INT	EGER ERRHANDLER, IERROR		28
Crea	ates an error handler that can	be attached to a file object. The user routine should	29
be, in C,	a function of type MPI_File_e	rrhandler_function, which is defined as	30
typedef	void MPI_File_errhandler	<pre>r_function(MPI_File *, int *,);</pre>	31
The	first argument is the file in u	se, the second is the error code to be returned.	32
	<u> </u>	e user routine file_errhandler_fn should be of the form:	33 34
	-	e user routilie me_ernandier_in should be of the form.	35
	T INTERFACE		36
		<pre>r_function(file, error_code) BIND(C)</pre>	37
	YPE(MPI_File) :: file NTEGER :: error_code		38
±.			39
With the	e Fortran mpi module and mpi:	f.h, the user routine FILE_ERRHANDLER_FN should	40
be of the	e form:		41 42
SUBROUT	INE FILE_ERRHANDLER_FUNCT	ION(FILE, ERROR CODE)	42
	EGER FILE, ERROR_CODE		44
			45
			46
			47
			48

```
1
     MPI_FILE_SET_ERRHANDLER(file, errhandler)
\mathbf{2}
       INOUT
                 file
                                            file (handle)
3
       IN
                 errhandler
                                            new error handler for file (handle)
4
5
6
     int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
\overline{7}
     MPI_File_set_errhandler(file, errhandler, ierror) BIND(C)
8
          TYPE(MPI_File), INTENT(IN) :: file
9
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
          INTEGER FILE, ERRHANDLER, IERROR
13
14
          Attaches a new error handler to a file. The error handler must be either a predefined
15
     error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
16
17
18
     MPI_FILE_GET_ERRHANDLER(file, errhandler)
19
       IN
                 file
                                            file (handle)
20
       OUT
                 errhandler
                                            error handler currently associated with file (handle)
21
22
23
     int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
^{24}
     MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)
25
          TYPE(MPI_File), INTENT(IN) :: file
26
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
30
          INTEGER FILE, ERRHANDLER, IERROR
^{31}
         Retrieves the error handler currently associated with a file.
32
33
     8.3.4 Freeing Errorhandlers and Retrieving Error Strings
34
35
36
37
     MPI_ERRHANDLER_FREE( errhandler )
38
       INOUT
                 errhandler
                                            MPI error handler (handle)
39
40
     int MPI_Errhandler_free(MPI_Errhandler *errhandler)
41
42
     MPI_Errhandler_free(errhandler, ierror) BIND(C)
43
          TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
46
          INTEGER ERRHANDLER, IERROR
47
48
```

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI\_ERRHANDLER\_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

			5
MPI_ERRC	<pre>DR_STRING( errorcode, string,</pre>	resultlen )	6
IN	errorcode	Error code returned by an MPI routine	7
OUT	string	Text that corresponds to the errorcode	8 9
OUT	resultlen	Length (in printable characters) of the result returned	10
		in string	11
			12
int MPI_E	rror_string(int errorcode	e, char *string, int *resultlen)	13
MDT Frror	string(errorcode string	g, resultlen, ierror) BIND(C)	14
	ER, INTENT(IN) :: error	-	15
	•		16
		<pre>IRING), INTENT(OUT) :: string</pre>	17
	ER, INTENT(OUT) :: resul		18
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	19
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)		20	
INTEG	ER ERRORCODE, RESULTLEN,	IERROR	21
CHARA	CTER*(*) STRING		22
Rotur	ng the error string associated	with an error code or class. The argument string	23

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI\_MAX\_ERROR\_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

*Rationale.* The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI\_ERROR\_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER\*(\*) can not be referenced in, for example, a PRINT statement. (End of rationale.)

#### 8.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI\_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI\_ERROR\_STRING).

To make it possible for an application to interpret an error code, the routine MPI\_ERROR\_CLASS converts any error code into one of a small set of standard error codes, called error classes. Valid error classes are shown in Table 8.1 and Table 8.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI\_ERROR\_STRING can be used to compute the error string associated with an error class. An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

$$0 = MPI_SUCCESS < MPI_ERR_... \le MPI_ERR_LASTCODE.$$

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

 $^{24}$ 

25

2627

28

29

30

31

32

33 34

3536

37

38

39

40

41

42

43

44

45

46

1		
2		N
3		No error
4		Invalid buffer pointer
5		Invalid count argument
6		Invalid datatype argument
7		Invalid tag argument
8		Invalid communicator
9		Invalid rank
10		Invalid request (handle)
11		Invalid root
12		Invalid group
13		Invalid operation
14		Invalid topology
15	, MPI_ERR_DIMS	Invalid dimension argument
16	, MPI_ERR_ARG	Invalid argument of some other kind
17	, MPI_ERR_UNKNOWN	Unknown error
18	MPI_ERR_TRUNCATE	Message truncated on receive
19	, MPI_ERR_OTHER	Known error not in this list
20	MPI_ERR_INTERN	Internal MPI (implementation) error
21	MPI_ERR_IN_STATUS	Error code is in status
22	MPI_ERR_PENDING	Pending request
23	3 MPI_ERR_KEYVAL	Invalid keyval has been passed
24	MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory
25	5	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	2	MPI_COMM_CONNECT
33	3 MPI_ERR_SERVICE	Invalid service name passed to
34	1	MPI_UNPUBLISH_NAME
35	, MPI_ERR_NAME	Invalid service name passed to
36	3	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
41	MPI_ERR_LOCKTYPE	Invalid locktype argument
42	2 MPI_ERR_ASSERT	Invalid assert argument
43	3 MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
44	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
45	5	
46		1. Ennon classes (Dont 1)
47	Table 8.	1: Error classes (Part 1)
48	3	

		2
MPI_ERR_RMA_RANGE		3
	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 7
	get memory is not attached)	8
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because	9
	of resource exhaustion)	9 10
MPI_ERR_RMA_SHARED		10
	memory cannot be shared (e.g., some pro-	11
	cess in the group of the specified commu-	12
	nicator cannot expose shared memory)	14
MPI_ERR_RMA_WRONG_FLAVOR	passed window has the wrong flavor for the	15
	called function	16
MPI_ERR_FILE	Invalid file handle	17
 MPI_ERR_NOT_SAME	Collective argument not identical on all	18
	processes, or collective routines called in	19
	a different order by different processes	20
MPI_ERR_AMODE	Error related to the amode passed to	21
	MPI_FILE_OPEN	22
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	23
	MPI_FILE_SET_VIEW	24
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	25
	a file which supports sequential access only	26
MPI_ERR_NO_SUCH_FILE	File does not exist	27
MPI_ERR_FILE_EXISTS	File exists	28
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	29
MPI_ERR_ACCESS	Permission denied	30
MPI_ERR_NO_SPACE	Not enough space	31
MPI_ERR_QUOTA	Quota exceeded	32
MPI_ERR_READ_ONLY	Read-only file or file system	33
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	34
	the file is currently open by some process	35
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	36
	tered because a data representation identi-	37
	fier that was already defined was passed to	38
	MPI_REGISTER_DATAREP	39
MPI_ERR_CONVERSION	An error occurred in a user supplied data	40
	conversion function.	41
MPI_ERR_IO	Other I/O error	42
MPI_ERR_LASTCODE	Last error code	43
		44
Table 8.2: Err	or classes (Part 2)	45
		46

Unofficial Draft for Comment Only

```
352
                                   CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT
1
           Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that
\mathbf{2}
           MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.
3
           Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the sepa-
4
           ration of error classes and error codes allows us to define the error classes this way.
5
           Having a known LASTCODE is often a nice sanity check as well. (End of rationale.)
6
7
8
9
     MPI_ERROR_CLASS( errorcode, errorclass )
10
       IN
                 errorcode
                                              Error code returned by an MPI routine
11
       OUT
                 errorclass
                                              Error class associated with errorcode
12
13
     int MPI_Error_class(int errorcode, int *errorclass)
14
15
     MPI_Error_class(errorcode, errorclass, ierror) BIND(C)
16
          INTEGER, INTENT(IN) :: errorcode
17
          INTEGER, INTENT(OUT) ::
                                       errorclass
18
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
19
     MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
20
          INTEGER ERRORCODE, ERRORCLASS, IERROR
21
22
          The function MPI_ERROR_CLASS maps each standard error code (error class) onto
23
     itself.
24
25
     8.5
            Error Classes, Error Codes, and Error Handlers
26
27
     Users may want to write a layered library on top of an existing MPI implementation, and
28
     this library may have its own set of error codes and classes. An example of such a library
29
     is an I/O library based on MPI, see Chapter 13 on page 491. For this purpose, functions
30
     are needed to:
^{31}
32
        1. add a new error class to the ones an MPI implementation already knows.
33
34
        2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
35
        3. associate strings with these error codes, so that MPI_ERROR_STRING works.
36
37
        4. invoke the error handler associated with a communicator, window, or object.
38
39
     Several functions are provided to do this. They are all local. No functions are provided
40
     to free error classes or codes: it is not expected that an application will generate them in
41
     significant numbers.
42
43
44
     MPI_ADD_ERROR_CLASS(errorclass)
45
```

```
46 OUT errorclass value for the new error class (integer)
47
48 int MPI_Add_error_class(int *errorclass)
```

<pre>MPI_Add_error_class(errorclass, ierror) BIND(C)</pre>	1
INTEGER, INTENT(OUT) :: errorclass	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
	4
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)	5
INTEGER ERRORCLASS, IERROR	6
Creates a new error class and returns the value for it.	7
	8
Rationale. To avoid conflicts with existing error codes and classes, the value is set	9
by the implementation and not by the user. $(End of rationale.)$	10
	11
Advice to implementors. A high-quality implementation will return the value for	12
a new errorclass in the same deterministic way on all processes. (End of advice to	13
implementors.)	14
Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass	15 16
may not be returned on all processes that make this call. Thus, it is not safe to assume	16
that registering a new error on a set of processes at the same time will yield the same	18
errorclass on all of the processes. However, if an implementation returns the new	19
errorclass in a deterministic way, and they are always generated in the same order on	20
the same set of processes (for example, all processes), then the value will be the same.	20
However, even if a deterministic algorithm is used, the value can vary across processes.	22
This can happen, for example, if different but overlapping groups of processes make	23
a series of calls. As a result of these issues, getting the "same" error on multiple	24
processes may not cause the same value of error code to be generated. (End of advice	25
to users.)	26
	27
The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-	28
defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is	29
associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the	30
current maximum error class including the user-defined ones. This is a local value and may	31
be different on different processes. The value returned by this key is always greater than or	32
equal to MPI_ERR_LASTCODE.	33

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.*)

42 MPI\_ADD\_ERROR\_CODE(errorclass, errorcode) IN errorclass error class (integer) OUT errorcode new error code to associated with errorclass (integer) 46 47 int MPI\_Add\_error\_code(int errorclass, int \*errorcode) 48

34

35

36

37

38

39

1 2 3 4 5	<pre>MPI_Add_error_code(errorclass, errorcode, ierror) BIND(C) INTEGER, INTENT(IN) :: errorclass INTEGER, INTENT(OUT) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
6 7	MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR
8 9	Creates new error code associated with errorclass and returns its value in errorcode.
10 11 12	<i>Rationale.</i> 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. ( <i>End of rationale.</i> )
12 13 14 15 16 17	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.)
18 19	MPI_ADD_ERROR_STRING(errorcode, string)
20	IN error code or class (integer)
21 22	IN string text corresponding to errorcode (string)
23 24	<pre>int MPI_Add_error_string(int errorcode, const char *string)</pre>
25 26 27 28	<pre>MPI_Add_error_string(errorcode, string, ierror) BIND(C) INTEGER, INTENT(IN) :: errorcode CHARACTER(LEN=*), INTENT(IN) :: string INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
29 30 31 32	MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING
33 34 35 36 37	Associates an error string with an error code or class. The string must be no more than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the calling language. The length of the string does not include the null terminator in C. Trailing blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that already has a string will replace the old string with the new string. It is erroneous to call
38 39 40	MPI_ADD_ERROR_STRING for an error code or class with a value $\leq$ MPI_ERR_LASTCODE. If MPI_ERROR_STRING is called when no string has been set, it will return a empty string (all spaces in Fortran, "" in C).
41 42 43 44	Section 8.3 on page 342 describes the methods for creating and associating error han- dlers with communicators, files, and windows.
45 46 47	
48	

1 MPI\_COMM\_CALL\_ERRHANDLER (comm, errorcode) 2 IN comm communicator with error handler (handle) 3 IN errorcode error code (integer) 4 5 int MPI\_Comm\_call\_errhandler(MPI\_Comm comm, int errorcode) 6 7 MPI\_Comm\_call\_errhandler(comm, errorcode, ierror) BIND(C) TYPE(MPI\_Comm), INTENT(IN) :: comm 9 INTEGER, INTENT(IN) :: errorcode 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI\_COMM\_CALL\_ERRHANDLER(COMM, ERRORCODE, IERROR) 12INTEGER COMM, ERRORCODE, IERROR 13 14This function invokes the error handler assigned to the communicator with the error 15code supplied. This function returns MPI\_SUCCESS in C and the same value in IERROR if 16 the error handler was successfully called (assuming the process is not aborted and the error 17 handler returns). 18 19 Users should note that the default error handler is Advice to users. 20MPI\_ERRORS\_ARE\_FATAL. Thus, calling MPI\_COMM\_CALL\_ERRHANDLER will abort 21the comm processes if the default error handler has not been changed for this com-22 municator or on the parent before the communicator was created. (End of advice to 23users.)  $^{24}$ 2526MPI\_WIN\_CALL\_ERRHANDLER (win, errorcode) 2728IN win window with error handler (handle) 29 IN errorcode error code (integer) 30 31int MPI\_Win\_call\_errhandler(MPI\_Win win, int errorcode) 32 33 MPI\_Win\_call\_errhandler(win, errorcode, ierror) BIND(C) 34 TYPE(MPI\_Win), INTENT(IN) :: win 35INTEGER, INTENT(IN) :: errorcode 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI\_WIN\_CALL\_ERRHANDLER(WIN, ERRORCODE, IERROR) 38 INTEGER WIN, ERRORCODE, IERROR 39 40 This function invokes the error handler assigned to the window with the error code 41

This function invokes the error handler assigned to the window with the error code supplied. 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).

Advice to users. As with communicators, the default error handler for windows is MPI\_ERRORS\_ARE\_FATAL. (End of advice to users.)

464748

42

43

44 45

```
1
     MPI_FILE_CALL_ERRHANDLER (fh, errorcode)
2
       IN
                 fh
                                             file with error handler (handle)
3
       IN
                 errorcode
                                             error code (integer)
4
5
6
     int MPI_File_call_errhandler(MPI_File fh, int errorcode)
7
     MPI_File_call_errhandler(fh, errorcode, ierror) BIND(C)
8
          TYPE(MPI_File), INTENT(IN) ::
                                             fh
9
          INTEGER, INTENT(IN) :: errorcode
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
11
12
     MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
13
          INTEGER FH, ERRORCODE, IERROR
14
          This function invokes the error handler assigned to the file with the error code supplied.
15
     This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler
16
     was successfully called (assuming the process is not aborted and the error handler returns).
17
18
           Advice to users. Unlike errors on communicators and windows, the default behavior
19
           for files is to have MPI_ERRORS_RETURN. (End of advice to users.)
20
21
           Advice to users.
                             Users are warned that handlers should not be called recursively
22
           with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or
23
           MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite
24
           recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER,
25
           MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER is called inside
26
           an error handler.
27
           Error codes and classes are associated with a process. As a result, they may be used
28
           in any error handler. Error handlers should be prepared to deal with any error code
29
           they are given. Furthermore, it is good practice to only call an error handler with the
30
           appropriate error codes. For example, file errors would normally be sent to the file
31
           error handler. (End of advice to users.)
32
33
34
            Timers and Synchronization
     8.6
35
36
     MPI defines a timer. A timer is specified even though it is not "message-passing," because
37
     timing parallel programs is important in "performance debugging" and because existing
38
     timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either incon-
     venient or do not provide adequate access to high-resolution timers. See also Section 2.6.4
39
40
     on page 19.
41
42
     MPI_WTIME()
43
44
     double MPI_Wtime(void)
45
46
     DOUBLE PRECISION MPI_Wtime() BIND(C)
47
     DOUBLE PRECISION MPI_WTIME()
48
```

MPI\_WTIME returns a floating-point number of seconds, representing elapsed wallclock 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:

```
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    .... stuff to be timed ...
    endtime = MPI_Wtime();
    printf("That took %f seconds\n",endtime-starttime);
}
```

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).

```
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()
 44

 int MPI\_Init(int \*argc, char \*((\*argv)[]))
 46

 MPI\_Init(ierror) BIND(C)
 47

Unofficial Draft for Comment Only

> $^{24}$

```
1
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
\mathbf{2}
     MPI_INIT(IERROR)
3
          INTEGER IERROR
4
5
          All MPI programs must contain exactly one call to an MPI initialization routine:
6
     MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erro-
7
      neous. The only MPI functions that may be invoked before the MPI initialization routines
8
      are called are MPI_GET_VERSION, MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED,
9
      MPI_FINALIZED, and any function with the prefix MPI_T_ (within the constraints for func-
10
      tions with this prefix listed in Section 14.3.4). The version for ISO C accepts the argc and
11
      argv that are provided by the arguments to main or NULL:
12
      int main(int argc, char *argv[])
13
      {
14
          MPI_Init(&argc, &argv);
15
16
          /* parse arguments */
17
          /* main program
                                 */
18
19
          MPI_Finalize();
                                 /* see below */
20
          return 0;
21
      }
22
23
      The Fortran version takes only IERROR.
24
          Conforming implementations of MPI are required to allow applications to pass NULL
25
      for both the argc and argv arguments of main in C.
26
          After MPI is initialized, the application can access information about the execution
27
      environment by querying the predefined info object MPI_INFO_ENV. The following keys are
28
      predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of
29
     mpiexec:
30
^{31}
      command name of program executed
32
      argv (space separated) arguments to command
33
34
      maxprocs Maximum number of MPI processes to start.
35
36
     soft Allowed values for number of processors
37
     host Hostname.
38
39
      arch Architecture name.
40
41
     wdir Working directory of the MPI process
42
      file Value is the name of a file in which additional information is specified.
43
44
      thread_level Requested level of thread support, if requested before the program started exe-
45
           cution.
46
47
48
```

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. Implementations 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.

**Example 8.4** If MPI is started with a call to

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

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)

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. (*End of advice to users.*)

Advice to implementors. High-quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (End of advice to implementors.)

```
MPI_FINALIZE()
```

```
int MPI_Finalize(void)
MPI_Finalize(ierror) BIND(C)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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 operations.

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

```
1
          The call to MPI_FINALIZE does not free objects created by MPI calls; these objects are
\mathbf{2}
     freed using MPI_xxx_FREE calls.
3
          MPI_FINALIZE is collective over all connected processes. If no processes were spawned,
4
     accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective
\mathbf{5}
     over the union of all processes that have been and continue to be connected, as explained
6
     in Section 10.5.4 on page 399.
7
          The following examples illustrates these rules
8
     Example 8.5 The following code is correct
9
10
              Process 0
                                           Process 1
11
               _____
                                           _____
12
              MPI_Init();
                                           MPI_Init();
13
              MPI_Send(dest=1);
                                           MPI_Recv(src=0);
14
                                           MPI_Finalize();
              MPI_Finalize();
15
16
17
     Example 8.6 Without a matching receive, the program is erroneous
18
19
              Process 0
                                           Process 1
               _____
                                            _____
20
              MPI_Init();
                                           MPI_Init();
21
              MPI_Send (dest=1);
22
23
              MPI_Finalize();
                                           MPI_Finalize();
^{24}
25
     Example 8.7 This program is correct: Process 0 calls MPI_Finalize after it has executed
26
     the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call
27
     that completes the matching receive operation before it calls MPI_Finalize.
28
29
       Process 0
                                          Proces 1
30
        _____
                                          _____
31
       MPI_Init();
                                          MPI_Init();
32
       MPI_Isend(dest=1);
                                          MPI_Recv(src=0);
33
       MPI_Request_free();
                                          MPI_Finalize();
34
       MPI_Finalize();
                                          exit();
35
     exit();
36
37
     Example 8.8 This program is correct. The attached buffer is a resource allocated by the
38
     user, not by MPI; it is available to the user after MPI is finalized.
39
40
         Process 0
                                           Process 1
41
         _____
                                            _____
42
         MPI_Init();
                                          MPI_Init();
43
         buffer = malloc(1000000);
                                          MPI_Recv(src=0);
44
         MPI_Buffer_attach();
                                          MPI_Finalize();
45
         MPI_Send(dest=1));
                                          exit();
46
         MPI_Finalize();
47
         free(buffer);
48
         exit();
```

**Example 8.9** This program is correct. The cancel operation must succeed, since the send cannot complete normally. The wait operation, after the call to MPI\_Cancel, is local – no matching MPI call is required on process 1.

Process O	Process 1
<pre>MPI_Issend(dest=1);</pre>	<pre>MPI_Finalize();</pre>
<pre>MPI_Cancel();</pre>	
<pre>MPI_Wait();</pre>	
<pre>MPI_Finalize();</pre>	

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 buffer; 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.*)

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, they 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);
```

 $^{31}$ 

```
1
     MPI_INITIALIZED( flag )
2
       OUT
                 flag
                                              Flag is true if MPI_INIT has been called and false
3
                                              otherwise.
4
5
     int MPI_Initialized(int *flag)
6
\overline{7}
     MPI_Initialized(flag, ierror) BIND(C)
8
          LOGICAL, INTENT(OUT) :: flag
9
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_INITIALIZED(FLAG, IERROR)
11
          LOGICAL FLAG
12
          INTEGER IERROR
13
14
          This routine may be used to determine whether MPI_INIT has been called.
15
     MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether
16
     MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one
17
     of the few routines that may be called before MPI_INIT is called.
18
19
     MPI_ABORT( comm, errorcode )
20
21
       IN
                 comm
                                              communicator of tasks to abort
22
       IN
                 errorcode
                                              error code to return to invoking environment
23
^{24}
     int MPI_Abort(MPI_Comm comm, int errorcode)
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_ABORT(COMM, ERRORCODE, IERROR)
^{31}
          INTEGER COMM, ERRORCODE, IERROR
32
33
          This routine makes a "best attempt" to abort all tasks in the group of comm. This
34
     function does not require that the invoking environment take any action with the error
35
     code. However, a Unix or POSIX environment should handle this as a return errorcode
36
     from the main program.
37
          It may not be possible for an MPI implementation to abort only the processes repre-
38
     sented by comm if this is a subset of the processes. In this case, the MPI implementation
39
     should attempt to abort all the connected processes but should not abort any unconnected
40
     processes. If no processes were spawned, accepted or connected then this has the effect of
41
     aborting all the processes associated with MPI_COMM_WORLD.
42
43
           Rationale. The communicator argument is provided to allow for future extensions of
44
           MPI to environments with, for example, dynamic process management. In particular,
45
           it allows but does not require an MPI implementation to abort a subset of
46
           MPI_COMM_WORLD. (End of rationale.)
47
48
```

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

# 8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI\_COMM\_SELF with a callback function. When MPI\_FINALIZE is called, it will first execute the equivalent of an MPI\_COMM\_FREE on MPI\_COMM\_SELF. This will cause the delete callback function to be executed on all keys associated with MPI\_COMM\_SELF, in the reverse order that they were set on MPI\_COMM\_SELF. If no key has been attached to MPI\_COMM\_SELF, then no callback is invoked. The "freeing" of MPI\_COMM\_SELF occurs before any other parts 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
```

 $\mathbf{2}$ 

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.)

13 14 15

16 17

18

1

2

3

4

5 6

7

8 9

10 11

12

# 8.8 Portable MPI Process Startup

A number of implementations of  $\mathsf{MPI}$  provide a startup command for  $\mathsf{MPI}$  programs that is of the form

19 20 21

# 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.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of mpirun MPI specifies mpiexec.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an mpiexec startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called mpiexec, it must be of the form described below.

It is suggested that

mpiexec -n <numprocs> <program>

be at least one way to start <program> with an initial MPI\_COMM\_WORLD whose group
 contains <numprocs> processes. Other arguments to mpiexec may be implementation dependent.

43

37 38

39

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).

<sup>48</sup> Analogous to MPI\_COMM\_SPAWN, we have

mpiexec -n	<maxproc< td=""><td>:s&gt;</td><td>1</td></maxproc<>	:s>	1
-soft	< -	>	2
-host	<	>	3
-arch	<	>	4
-wdir	<	>	5
-path	<	>	6
-file	<	>	7
			8
<command line=""/>			9
			10
for the case where a s	ingle comn	nand line for the application program and its arguments	11
will suffice. See Section 10.3.4 for the meanings of these arguments. For the case			12
		SPAWN_MULTIPLE there are two possible formats:	13
Form A:		-	14
ronn A.			15
mniowaa ( Kab		nents> } : { } : { } : : { }	16
mpiexec i vab	ove argum	lenus / ; ; ; / ; ; ; ; ; ; ; /	17
			18
As with MPI_COMM_SPAWN, all the arguments are optional. (Even the -n x argu-			19
		implementation dependent. It might be 1, it might be	20
taken from an environment variable, or it might be specified at compile time.) The			21
names and meanings of the arguments are taken from the keys in the info argument			22
to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments			23
as well.			24
Note that Form A, though convenient to type, prevents colons from being program			25
arguments. Therefor	e an alterr	nate, file-based form is allowed:	26
Form B:			27
			28
<pre>mpiexec -configfile <filename></filename></pre>			29
1	0		30
where the lines of <i>&lt;</i> 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 '\'.			31 32
			32
			33 34
···· P ··· ···· ····			35
Example 8.11 Sta	rt 16 insta	nces of myprog on the current or default machine:	36
Example 6.11 Start to instances of myprog on the current of default machine.			37
mpiexec -n 16 myprog			38
- <b>r</b>			39
			40
<b>Example 8.12</b> Start 10 processes on the machine called ferrari:			41
			42
mpiexec -n 10 -host ferrari myprog			43
			44
Example 9.19 $O_{\pm -}$	nt three co	ning of the same program with different commend line	45
Example 8.13 Sta	rt three co	pies of the same program with different command-line	

Example 8.13 Start three copies of the same program with different command-line arguments:

mpiexec myprog infile1 : myprog infile2 : myprog infile3

**Unofficial Draft for Comment Only** 

46

1	<b>Example 8.14</b> Start the ocean program on five Suns and the atmos program on 10	
2	RS/6000's:	
3		
4	mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos	
5		
6	It is assumed that the implementation in this case has a method for choosing hosts of	
7	the appropriate type. Their ranks are in the order specified.	
8		
9	<b>Example 8.15</b> Start the ocean program on five Suns and the atmos program on 10	
10	RS/6000's (Form B):	
11		
12	mpiexec -configfile myfile	
13		
14	where myfile contains	
15		
16	-n 5 -arch sun ocean	
17	-n 10 -arch rs6000 atmos	
18		
19	(End of advice to implementors.)	
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 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 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 

 $^{24}$ 

 $^{31}$ 

1include 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 7automatically 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 11 MPI\_INFO\_CREATE(info) 1213 OUT 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 22 MPI\_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 info object (handle) 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 40 CHARACTER\*(\*) 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 46raised, respectively.

MPI_INFO_DELETE(info, key) <sup>1</sup>					
INOUT	info	info object (handle)	2		
IN	key	key (string)	3 4		
			5		
int MPI_]	Info_delete(MPI_Info info	, const char *key)	6		
		·	7		
	_delete(info, key, ierror) (MPI_Info), INTENT(IN) ::		8		
	ACTER(LEN=*), INTENT(IN)		9		
	ER, OPTIONAL, INTENT(OUT)		10 11		
	_DELETE(INFO, KEY, IERROR)		11		
	ER INFO, IERROR		13		
	ACTER*(*) KEY		14		
MDL	NEO DELETE deletes a (kov	value) pair from info. If key is not defined in info	15		
	ises an error of class MPI_ERR	value) pair from info. If key is not defined in info, INFO NOKEY	16		
the can ra			17		
			18 19		
	_GET(info, key, valuelen, value	e, flag)	20		
IN	info	info object (handle)	21		
IN	key	key (string)	22		
IN	valuelen	length of value arg (integer)	23		
OUT	value	value (string)	24 25		
OUT	flag	true if key defined, false if not (boolean)	23 26		
001			27		
int MPI_]	Info_get(MPI_Info info, co	onst char *key, int valuelen, char *value,	28		
_	int *flag)		29		
MPT Info	get(info key valuelen	value, flag, ierror) BIND(C)	30		
	(MPI_Info), INTENT(IN) ::	-	31 32		
	ACTER(LEN=*), INTENT(IN)		33		
INTEC	SER, INTENT(IN) :: value	len	34		
	ACTER(LEN=valuelen), INTE	NT(OUT) :: value	35		
	CAL, INTENT(OUT) :: flag	· · · · · · · · · · · · · · · · · · ·	36		
INTEC	GER, OPTIONAL, INTENT(OUT)	) :: ierror	37		
MPI_INFO_	GET(INFO, KEY, VALUELEN,	VALUE, FLAG, IERROR)	38		
	GER INFO, VALUELEN, IERRO	R	39 40		
	ACTER*(*) KEY, VALUE CAL FLAG		41		
			42		
		ssociated 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, otherwise it sets flag to false and leaves value unchanged, valuelen is the number of characters					

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. 44 45 46 47 48

### Unofficial Draft for Comment Only

```
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
       IN
                 info
                                              info object (handle)
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)		1
IN	info	info object (handle)	2
IN	n	key number (integer)	3
			4 5
OUT	key	key (string)	6
· . NDT T			7
int MPI_I	nfo_get_nthkey(MPI_Info i	nio, int n, char *key)	8
MPI_Info_	get_nthkey(info, n, key,	ierror) BIND(C)	9
	<pre>MPI_Info), INTENT(IN) ::</pre>	info	10
	ER, INTENT(IN) :: n		11
	CTER(LEN=*), INTENT(OUT)		12
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	13
MPI_INFO_	GET_NTHKEY(INFO, N, KEY,	IERROR)	14 15
	ER INFO, N, IERROR		16
CHARA	CTER*(*) KEY		17
This fu	unction returns the nth define	d key in info. Keys are numbered $0 \dots N - 1$ where	18
N is the v	alue returned by $MPI_{INFO}_{INFO}$	<b>GET_NKEYS</b> . All keys between 0 and $N - 1$ are	19
0		f a given key does not change as long as info is not	20
modified w	ith MPI_INFO_SET or MPI_II	NFO_DELETE.	21
			22
MPI_INFO	_DUP(info, newinfo)		23
IN	info	info object (handle)	24 25
			26
OUT	newinfo	info object (handle)	27
tert MDT T			28
int MPI_I	nfo_dup(MPI_Info info, MF	'l_inio *newinio)	29
MPI_Info_	dup(info, newinfo, ierror	) BIND(C)	30
	MPI_Info), INTENT(IN) ::		31
	MPI_Info), INTENT(OUT) ::		32
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	33 34
MPI_INFO_	DUP(INFO, NEWINFO, IERROF	a)	35
INTEG	ER INFO, NEWINFO, IERROR		36
MPI II	NFO DUP duplicates an exis	ting info object, creating a new object, with the	37
	value) pairs and the same orde		38
	, -		39
	FDEE(infa)		40
	_FREE(info)		41
INOUT	info	info object (handle)	42
			43 44
int MPI_I	nfo_free(MPI_Info *info)		45
MPI_Info_	free(info, ierror) BIND(C	;)	46
	MPI_Info), INTENT(INOUT)		47
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	48

Unofficial Draft for Comment Only

1	MPI_INFO_FREE(INFO, IERROR)
2	INTEGER INFO, IERROR
3	This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is
4	interpreted each time the info is passed to a routine. Changes to an info after return from
5	a routine do not affect that interpretation.
6	
7	
8	
9 10	
10	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25 26	
20	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38 39	
39 40	
40	
42	
43	
44	
45	
46	
47	
48	

## Chapter 10

# **Process Creation and Management**

### 10.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allow for a variety of approaches to process management while placing minimal restrictions on the execution environment.

The MPI model for process creation allows both the creation of an initial set of processes related by their membership in a common MPI\_COMM\_WORLD and the creation and management of processes after an MPI application has been started. A major impetus for the later form of process creation comes from the PVM [24] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. 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

 $^{24}$ 

 $^{31}$ 

clean interface between an application and system software.

- MPI must guarantee communication determinism in the presense of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.

The process management model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI maintains a consistent concept of a communicator, regardless of how its members came into existence. A communicator is never changed once created, and it is always created using deterministic collective operations.

10.2 The Dynamic Process Model

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 that 37 there is an enormous range of runtime environments and application requirements, and MPI 38 39 must not be tailored to any particular one. Examples of such environments are:

40 41

42

43

44

45

46

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.
- 47 48

10

11

12

13

1415

16

18

- 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.

375

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10 11

12

13

14

15

16 17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41 42

43

44

45

46

47 48

#### **Unofficial Draft for Comment Only**

1 2 3 4 5 6	MPI_ name of M	COMM_WORLD tells a progra	(See Section 10.5.1 on page 397) on m how "large" the initial runtime environment is, isefully be started in all. One can subtract the size alue to find out how many processes might usefully eady running.			
7 8	10.3 Process Manager Interface					
9 10	10.3.1 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 S	tarting Processes and Establis	shing Communication			
16 17 18		ing routine starts a number of urning an intercommunicator.	MPI processes and establishes communication with			
19 20 21 22 23 24 25 26	Advice to users. It is possible in MPI to start a static SPMD or MPMD appli- cation by starting first 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, argv, m array_of_errcodes)	naxprocs, 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 $command$ (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)			
40 47 48	OUT	array_of_errcodes	one code per process (array of integer)			

```
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

#### Unofficial Draft for Comment Only

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
   that calls MPI\_INIT, the results are undefined. Implementations may allow this case to
   work but are not required to.
  - Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI\_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)
- 18 19

34

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI\_ARGV\_NULL may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as NULL.

```
27
28 Example 10.1 Examples of argv in C and Fortran
29 To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
30
31 char command[] = "ocean";
32 char *argv[] = {"-gridfile", "ocean1.grd", NULL};
33 MPI_Comm_spawn(command, argv, ...);
```

```
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
```

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 384 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 46 handle of type MPI\_Info in C and Fortran with the mpi\_f08 module and INTEGER in 47 Fortran with the mpi module or the include file mpif.h. It is a container for a number 48

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

 $\frac{44}{45}$ 

of user-specified (key,value) pairs. key and value are strings (null-terminated char\* in C,
 character\*(\*) in Fortran). Routines to create and manipulate the info argument are
 described in Section 9 on page 367.

For the SPAWN calls, info provides additional (and possibly implementation-dependent)
 instructions to MPI and the runtime system on how to start processes. An application may
 pass MPI\_INFO\_NULL in C or Fortran. Portable programs not requiring detailed control over
 process locations should use MPI\_INFO\_NULL.

<sup>8</sup> MPI does not specify the content of the info argument, except to reserve a number of <sup>9</sup> special key values (see Section 10.3.4 on page 384). The info argument is quite flexible and <sup>10</sup> could even be used, for example, to specify the executable and its command-line arguments. <sup>11</sup> In this case the command argument to MPI\_COMM\_SPAWN could be empty. The ability to <sup>12</sup> do this follows from the fact that MPI does not specify how an executable is found, and the <sup>13</sup> info argument can tell the runtime system where to "find" the executable "" (empty string). <sup>14</sup> Of course a program that does this will not be portable across MPI implementations.

15

The root argument All arguments before the root argument are examined only on the
 process whose rank in comm is equal to root. The value of these arguments on other
 processes is ignored.

19

33

34

39

20The 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,  $\operatorname{array_of}_{\operatorname{errcodes}}$  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 24the 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. In C++30 this constant does not exist, and the array\_of\_errcodes argument may be omitted from the  $^{31}$ argument list. 32

Advice to implementors. MPI\_ERRCODES\_IGNORE in Fortran is a special type of constant, like MPI\_BOTTOM. See the discussion in Section 2.5.4 on page 15. (End of advice to implementors.)

MPI\_COMM\_GET\_PARENT(parent)

OUT the parent communicator (handle) parent 4041 int MPI\_Comm\_get\_parent(MPI\_Comm \*parent) 4243MPI\_Comm\_get\_parent(parent, ierror) BIND(C) 44 TYPE(MPI\_Comm), INTENT(OUT) :: parent 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647MPI\_COMM\_GET\_PARENT(PARENT, IERROR) INTEGER PARENT, IERROR 48

If a process was started with MPI\_COMM\_SPAWN or MPI\_COMM\_SPAWN\_MULTIPLE, MPI\_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.

 $\mathbf{5}$ 

```
382
```

```
1
     MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs,
\mathbf{2}
                     array_of_info, root, comm, intercomm, array_of_errcodes)
3
       IN
                 count
                                             number of commands (positive integer, significant to
4
                                             MPI only at root — see advice to users)
5
       IN
                 array_of_commands
                                             programs to be executed (array of strings, significant
6
                                             only at root)
7
8
       IN
                                             arguments for commands (array of array of strings,
                 array_of_argv
9
                                             significant only at root)
10
       IN
                 array_of_maxprocs
                                             maximum number of processes to start for each com-
11
                                             mand (array of integer, significant only at root)
12
       IN
                 array_of_info
                                             info objects telling the runtime system where and how
13
                                             to start processes (array of handles, significant only at
14
                                             root)
15
16
       IN
                                             rank of process in which previous arguments are ex-
                 root
17
                                             amined (integer)
18
       IN
                                             intracommunicator containing group of spawning pro-
                 comm
19
                                             cesses (handle)
20
       OUT
                 intercomm
                                             intercommunicator between original group and newly
21
                                             spawned group (handle)
22
23
       OUT
                 array_of_errcodes
                                             one error code per process (array of integer)
^{24}
25
     int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
26
                     char **array_of_argv[], const int array_of_maxprocs[], const
27
                    MPI_Info array_of_info[], int root, MPI_Comm comm,
28
                    MPI_Comm *intercomm, int array_of_errcodes[])
29
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
30
                    array_of_maxprocs, array_of_info, root, comm, intercomm,
^{31}
                    array_of_errcodes, ierror) BIND(C)
32
          INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
33
          CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
34
          array_of_argv(count, *)
35
          TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Comm), INTENT(OUT) ::
                                               intercomm
38
          INTEGER :: array_of_errcodes(*)
39
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
40
41
     MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
42
                    ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
43
                    ARRAY_OF_ERRCODES, IERROR)
44
          INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
45
          INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
46
          CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
47
48
```

MPI\_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.)

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

```
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
8
     "atmos" with argument "atmos.grd" in C:
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'
              array_of_argv(1, 3) = ', '
23
^{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 Forman-50 triplets, 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 $a, a + c, a + 2c,, a + ck$ , where for $c > 0$ , k is the largest integer for which $a + ck \le b$ and for $c < 0$ , k is the largest integer for which $a + ck \ge b$ . If $b > a$ then c must be positive. If $b < a$ then c must be negative.	12 13 14 15
Examples:	16
1. <b>a:b</b> gives a range between $a$ and $b$	17 18
2. 0:N gives full "soft" functionality	19
3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows power-of-two number of processes.	20 21
4. 2:10000:2 allows even number of processes.	22
5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.	23 24
5. 2.10.2, 7 and $8. 2, 4, 0, 7, 6, 61 10$ processes.	24 25
10.3.5 Spawn Example	26
	27
Manager-worker Example Using MPI_COMM_SPAWN.	28
/* manager */	29
<pre>#include "mpi.h"</pre>	30
<pre>int main(int argc, char *argv[]) </pre>	31 32
{	33
<pre>int world_size, universe_size, *universe_sizep, flag; MPI_Comm everyone;</pre>	34
<pre>MPI_Comm everyone;</pre>	35
chai worker_program[100],	36
MPI_Init(&argc, &argv);	37
MPI_Comm_size(MPI_COMM_WORLD, &world_size);	38
In 1_00mm_0120(In 1_00In1_#01022), wwo114_0120),	39
if (world_size != 1) error("Top heavy with management");	40
	41
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,	42
&universe_sizep, &flag);	43
if (!flag) {	44
$printf("This MPI does not support UNIVERSE_SIZE. How many\n$	45
processes total?");	46
<pre>scanf("%d", &amp;universe_size);</pre>	47

} else universe\_size = \*universe\_sizep;

Unofficial Draft for Comment Only

 $1 \\ 2$ 

```
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.
42
         * The manager is represented as the process with rank 0 in (the remote
         * 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.

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

388

1

 $\mathbf{2}$ 

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.

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	port_name	newly obtablished port (burns)
46	int MDT		info show through none)
47	int MPI_	Upen_port(MP1_Inio	info, char *port_name)
48	MPI_Open	_port(info, port_n	ame, ierror) 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 MI	PI_Close_port(const	char *port_name)
	lose_port(port_name	
CI	HARACTER(LEN=*), INT	TENT(IN) :: port_name
II	NTEGER, OPTIONAL, IN	NTENT(OUT) :: ierror

#### **Unofficial Draft for Comment Only**

1

2

3

4

5

6

7 8

9

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 $^{24}$ 

25

26

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_COMMECT (port_name, mo				
IN	port_name	network address (string, used only on root)	2		
IN	info	implementation-dependent information (handle, used	3		
	into	only on root)	4		
		• /	5		
IN	root	rank in <b>comm</b> of root node (integer)	6		
IN	comm	intracommunicator over which call is collective (han-	7		
		dle)	8 9		
OUT	newcomm	intercommunicator with server as remote group (han-	9 10		
		dle)	10		
			11		
int MDT C	omm connect(const char *r	oort_name, MPI_Info info, int root,	12		
Int mI_C	MPI_Comm comm, MPI_Co		14		
			15		
	-	root, comm, newcomm, ierror) BIND(C)	16		
	CTER(LEN=*), INTENT(IN) :	1	17		
	MPI_Info), INTENT(IN) ::	info	18		
	ER, INTENT(IN) :: root		19		
	= , , , , , , , ,	comm	20		
	MPI_Comm), INTENT(OUT) ::		21		
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	22		
MPI COMM	CONNECT(PORT NAME, INFO.	ROOT, COMM, NEWCOMM, IERROR)	23		
	CTER*(*) PORT_NAME		24		
	ER INFO, ROOT, COMM, NEWC	COMM. IERROR	25		
			26		
		ation with a server specified by port_name. It is	27		
collective over the calling communicator and returns an intercommunicator in which the					

MPL CO	MM C	ONNECT(	nort	name	info	root	comm	newcomm	)
1011 - 00			port_	inanne,	mo,	1001,	comm,	newconnin	

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

#### Unofficial Draft for Comment Only

1

44 45

28

29

30

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

42

43

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

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 is, 8 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)			
20 21						
22	int MP		char *service_name, MPI_Info info, const			
23		char *port_nam	e)			
24	MPI_Pul	olish_name(service_name)	ne, info, port_name, ierror) BIND(C)			
25	TYPE(MPI_Info), INTENT(IN) :: info					
26 27	CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name					
28	TN.	TEGER, OPTIONAL, INTEN	VI(UUT) :: ierror			
29	MPI_PU	BLISH_NAME(SERVICE_NAM	HE, INFO, PORT_NAME, IERROR)			

30 INTEGER INFO, 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)
int MPI_U	npublish_name(const char char *port_name)	*service_name, MPI_Info info, const

```
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
```

```
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

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10 11

12

13

14

15

16

17

18

 $^{31}$ 

32

33

34 35

36

37

38

39

40

41

42

43

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). 22If 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. 27Reserved Key Values 2810.4.5 29The 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 43 not use service names at all. 44On the server side: 454647 char myport[MPI\_MAX\_PORT\_NAME]; 48 MPI\_Comm intercomm;

/* Мрт (	*/ Dpen_port(MPI_INFO_NULL, myport);
	<pre>spen_port(MP1_INF0_NOLL, myport); sf("port name is: %s\n", myport);</pre>
1	
	<pre>Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &amp;intercomm);</pre>
/* do	o something with intercomm */
The serve	r prints out the port name to the terminal and the user must type it in when
starting u	p the client (assuming the MPI implementation supports stdin such that this
works). O	n the client side:
MPI_0	Comm intercomm;
	<pre>name[MPI_MAX_PORT_NAME];</pre>
print	tf("enter port name: ");
gets	(name);
MPI_(	Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
Ocean/Atr	nosphere - Relies on Name Publishing
,	
In this example, the "ocean" application is the "server" side of a coupled ocea	
climate mo	odel. It assumes that the MPI implementation publishes names.
MPI (	<pre>Dpen_port(MPI_INFO_NULL, port_name);</pre>
	Publish_name("ocean", MPI_INFO_NULL, port_name);
MPI_C	<pre>Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &amp;intercomm);</pre>
/* do	o something with intercomm */
MPI_U	<pre>Inpublish_name("ocean", MPI_INFO_NULL, port_name);</pre>
On the cli	ent side
Jn the ch	ent side:
MPI_I	Lookup_name("ocean", MPI_INFO_NULL,
MPI_0	Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
	&intercomm);
Simple Clie	ent-Server Example.
This is a s	simple example; the server accepts only a single connection at a time and serves
	ection until the client requests to be disconnected. The server is a single process.
Here is the server. It accepts a single connection and then processes data un	
receives a	message with tag 1. A message with tag 0 tells the server to exit.
tinclude	"mpi b"
#include	
Int main {	(int argc, char *argv[])
	Comm client;
191 <b>-</b> 1 1	
	Status status;

**Unofficial Draft for Comment Only** 

```
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, 43 44some 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

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

5 6

7 8

9

10 11 12

13 14

15

16

17

18

19

20

21

30

3132

33 34

35

36

37 38

39

40 41

42

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

### 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.)
- <sup>38</sup> 10 F 3

30 31

32

33 34

35 36

37

39

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

## Unofficial Draft for Comment Only

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 don't share the same MPI\_COMM\_WORLD may become disconnected (independent) if the communication path between them is broken by using MPI\_COMM\_DISCONNECT.

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $44 \\ 45$ 

1	The following additional rules apply to MPI routines in other chapters:			
2 3	<ul> <li>MPI_FINALIZE is collective over a set of connected processes.</li> <li>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.</li> </ul>			
4 5 6 7 8				
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	int MPI_Comm_disconnect(MPI_Comm *comm)			
18 19 20 21	MPI_Comm_disconnect(comm, ierror) BIND(C) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
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 same as for MPI_FINALIZE.			
31 32 33 34	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. Notes 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 44	<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> )			
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)</pre>					
<pre>MPI_Comm_join(fd, intercomm, ierror) BIND(C)     INTEGER, INTENT(IN) :: fd</pre>					
TYPE(MPI_Comm), INTENT(OUT) :: intercomm					
INTEG	ER, 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 [46, 50]. 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).

#### Unofficial Draft for Comment Only

If MPI is unable to create an intercommunicator, but is able to leave the socket in its  $\mathbf{2}$ 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 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. MPI is free to use any available communication path(s) Advice to implementors. 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 399 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.

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.

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

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 7in 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 <sup>17</sup> process in which the memory is accessed. Thus, in a put operation, source=origin and <sup>18</sup> destination=target; in a get operation, source=target and destination=origin.

19 20

21

## 11.2 Initialization

22MPI provides the following window initialization functions, MPI\_WIN\_CREATE, 23MPI\_WIN\_ALLOCATE, MPI\_WIN\_ALLOCATE\_SHARED and MPI\_WIN\_CREATE\_DYNAMIC  $^{24}$ that are collective on an intracommunicator. MPI\_WIN\_CREATE allows each process to 25specify a "window" in its memory that is made accessible to accesses by remote processes. 26The call returns an opaque object that represents the group of processes that own and ac-27cess the set of windows, and the attributes of each window, as specified by the initialization 28call. MPI\_WIN\_ALLOCATE differs from MPI\_WIN\_CREATE in that the user does not pass 29allocated memory; MPI\_WIN\_ALLOCATE returns a pointer to memory allocated by the 30 MPI implementation. MPI\_WIN\_ALLOCATE\_SHARED differs from MPI\_WIN\_ALLOCATE  $^{31}$ in that the allocated memory can be accessed from all processes in the window's group with 32 direct load/store instructions. Some restrictions may apply to the specified communica-33 tor. MPI\_WIN\_CREATE\_DYNAMIC creates a window that allows the user to dynamically 34control which memory is exposed by the window. 35

11.2.1 Window Creation

37 38 39

36

# 40 MPI\_WIN\_CREATE(base, size, disp\_unit, info, comm, win)

42 43INsizesize of window in bytes (non-negative integer)44INdisp_unitlocal unit size for displacements, in bytes (positive in- teger)46INinfoinfo argument (handle)47484848	41	IN	base	initial address of window (choice)
<ul> <li>45 teger)</li> <li>46 IN info info argument (handle)</li> <li>47</li> </ul>	42 43	IN	size	size of window in bytes (non-negative integer)
47 Into argument (nandie)		IN	disp_unit	
48		IN	info	info argument (handle)
	48			

IN	comm	intra-communicator (handle)	1
OUT	win	window object returned by the call (handle)	2
			3
int MPI	_Win_create(void *base, MP]	I_Aint size, int disp_unit, MPI_Info info,	4
	MPI_Comm comm, MPI_W	-	5 6
NDT Uda		the information is a second DIND(Q)	7
	_create(base, size, disp_u E(*), DIMENSION(), ASYNC	nit, info, comm, win, ierror) BIND(C)	8
	EGER(KIND=MPI_ADDRESS_KIND)		9
	EGER, INTENT(IN) :: disp_1		10
	$E(MPI_Info), INTENT(IN) ::$		11
	E(MPI_Comm), INTENT(IN) ::		12
TYPI	E(MPI_Win), INTENT(OUT) ::	win	13
INT	EGER, OPTIONAL, INTENT(OUT)	) :: ierror	14
мрт штм	CREATE (BASE SIZE DISD III	NIT, INFO, COMM, WIN, IERROR)	15
	De> BASE(*)	vii, inro, com, win, innon,	16
	EGER(KIND=MPI_ADDRESS_KIND)	) STZE	17
	EGER DISP_UNIT, INFO, COMM		18 19
	_ , ,		19

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. In C and C++, base is the starting address of a memory region. In Fortran, one can pass 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 628). 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 later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (*End of advice to users.*)

The info argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info keys are predefined:

no\_locks — if set to true, then the implementation may assume that passive target synchronization (i.e., MPI\_WIN\_LOCK, MPI\_LOCK\_ALL) will not be used on the given
 window. This implies that this window is not used for 3-party communication, and
 RMA can be implemented with no (less) asynchronous agent activity at this process.

#### Unofficial Draft for Comment Only

 $^{24}$ 

accumulate\_ordering — controls the ordering of accumulate operations at the target. See Section 11.7.2 for details.

accumulate\_ops — if set to same\_op, the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation. If set to same\_op\_no\_op, then the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation or MPI\_NO\_OP. This can eliminate the need to protect access for certain operation types where the hardware can guarantee atomicity. The default is same\_op\_no\_op.

Advice to users. The info query mechanism described in Section 11.2.7 can be used to query the specified info arguments windows that have been passed to a library. It is recommended that libraries check attached info keys for each passed window. (*End* of advice to users.)

The various processes in the group of comm may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the get, put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to undefined results.

Rationale. The reason for specifying the memory that may be accessed from another process in an RMA operation is to permit the programmer to specify what memory can be a target of RMA operations and for the implementation to enforce that specification. For example, with this definition, a server process can safely allow a client process to use RMA operations, knowing that (under the assumption that the MPI implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (End of rationale.)

30 31 32

33

34

35

36

37

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI\_ALLOC\_MEM (Section 8.2, page 339) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (End of advice to users.)

Advice to implementors. In cases where RMA operations use different mechanisms 38 in different memory areas (e.g., load/store in a shared memory segment, and an asyn-39 chronous handler in private memory), the MPI\_WIN\_CREATE call needs to figure out 40 which type of memory is used for the window. To do so, MPI maintains, internally, the 41 list of memory segments allocated by MPI\_ALLOC\_MEM, or by other, implementa-42tion specific, mechanisms, together with information on the type of memory segment 43 allocated. When a call to MPI\_WIN\_CREATE occurs, then MPI checks which segment 44 contains each window, and decides, accordingly, which mechanism to use for RMA 45operations. 46

<sup>47</sup> Vendors may provide additional, implementation-specific mechanisms to allocate or
 <sup>48</sup> to specify memory regions that are preferable for use in one-sided communication. In

1

2

3

4

5

6

7

8

9 10 11

12

13

14

15

22

23

24

25

26

27

28

	particular, such mechanisms regions.	can be used to place static variables into such preferred	1 2
	Implementors should docum of advice to implementors.)	ent any performance impact of window alignment. ( $End$	3 4 5
11.2.	2 Window That Allocates	Memory	6 7 8 9
MPI_	WIN_ALLOCATE(size, disp_u	init, info, comm, baseptr, win)	10 11
IN	size	size of window in bytes (non-negative integer)	11
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	13 14
IN	info	info argument (handle)	15 16
IN	comm	intra-communicator (handle)	10
OU	T baseptr	initial address of window (choice)	18
OU	T win	window object returned by the call (handle)	19 20
int		t size, int disp_unit, MPI_Info info, void *baseptr, MPI_Win *win)	21 22 23
MPI_	Win_allocate(size, disp_	unit, info, comm, baseptr, win, ierror) BIND(C)	$^{24}$
	USE, INTRINSIC :: ISO_C		25
	INTEGER (KIND=MPI_ADDRESS	-	26 27
	INTEGER, INTENT(IN) :: TYPE(MPI_Info), INTENT(I	-	21
	TYPE(MPI_Comm), INTENT(I		29
	TYPE(C_PTR), INTENT(OUT)		30
	TYPE(MPI_Win), INTENT(OU		31
	INTEGER, OPTIONAL, INTEN	I(OUT) :: ierror	32
MPI_	WIN_ALLOCATE(SIZE, DISP_	UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	33 34
	INTEGER DISP_UNIT, INFO,		35
	INTEGER(KIND=MPI_ADDRESS	_KIND) SIZE, BASEPTR	36

This is a collective call executed by all processes in the group of comm. On each process, it allocates memory of at least size bytes, returns a pointer to it, and returns a window object that can be used by all processes in comm to perform RMA operations. The returned memory consists of size bytes local to each process, starting at address baseptr  $^{41}$ and is associated with the window as if the user called MPI\_WIN\_CREATE on existing memory. The size argument may be different at each process and size = 0 is valid; however, a library might allocate and expose more memory in order to create a fast, globally symmetric allocation. The discussion of and rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 8.2 also apply to MPI\_WIN\_ALLOCATE; in particular, see the rationale in Section 8.2 for an explanation of the type used for baseptr.

If the Fortran compiler provides TYPE(C\_PTR), then the following interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with 48

#### Unofficial Draft for Comment Only

```
1
     the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR,
\mathbf{2}
     but with a different linker name:
3
4
     INTERFACE MPI_WIN_ALLOCATE
          SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
5
                   WIN, IERROR)
6
               USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7
               INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
8
               INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
9
               TYPE(C_PTR) :: BASEPTR
10
          END SUBROUTINE
11
     END INTERFACE
12
13
          The linker name base of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The
14
     implied linker names are described in Section 17.1.5 on page 607.
15
16
           Rationale. By allocating (potentially aligned) memory instead of allowing the user
17
           to pass in an arbitrary buffer, this call can improve the performance for systems with
18
           remote direct memory access. This also permits the collective allocation of memory
19
           and supports what is sometimes called the "symmetric allocation" model that can be
20
           more scalable (for example, the implementation can arrange to return an address for
21
           the allocated memory that is the same on all processes). (End of rationale.)
22
23
          The info argument can be used to specify hints similar to the info argument for
24
     MPI_WIN_CREATE and MPI_ALLOC_MEM. The following info key is predefined:
25
26
      same_size — if set to true, then the implementation may assume that the argument size is
27
           identical on all processes.
28
29
30
     11.2.3 Window That Allocates Shared Memory
^{31}
32
33
     MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)
34
       IN
                 size
                                              size of local window in bytes (non-negative integer)
35
36
       IN
                 disp_unit
                                              local unit size for displacements, in bytes (positive in-
37
                                              teger)
38
       IN
                 info
                                              info argument (handle)
39
       IN
                 comm
                                              intra-communicator (handle)
40
41
       OUT
                 baseptr
                                              address of local allocated window segment (choice)
42
       OUT
                 win
                                              window object returned by the call (handle)
43
44
     int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
45
                     MPI_Comm comm, void *baseptr, MPI_Win *win)
46
47
     MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
48
                     BIND(C)
```

USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: size INTEGER, INTENT(IN) :: disp\_unit TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(C\_PTR), INTENT(OUT) :: baseptr TYPE(MPI\_Win), INTENT(OUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror WIN ALLOCATE SHARED(SIZE, DISP UNIT, INFO, COMM, BASEPTR, WIN, IERROR)

MPI\_WIN\_ALLOCATE\_SHARED(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR

13 This is a collective call executed by all processes in the group of comm. On each 14process i, it allocates memory of at least size bytes that is shared among all processes in 15comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used 16for load/store accesses on the calling process. The locally allocated memory can be the 17 target of load/store accesses by remote processes; the base pointers for other processes 18can be queried using the function MPI\_WIN\_SHARED\_QUERY. The call also returns a 19window object that can be used by all processes in comm to perform RMA operations. 20The size argument may be different at each process and size = 0 is valid. It is the user's 21responsibility to ensure that the communicator comm represents a group of processes that 22can create a shared memory segment that can be accessed by all processes in the group. 23The discussions of rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 8.2  $^{24}$ also apply to MPI\_WIN\_ALLOCATE\_SHARED; in particular, see the rationale in Section 8.2 25for an explanation of the type used for **baseptr**. The allocated memory is contiguous across 26process ranks unless the info key alloc\_shared\_noncontig is specified. Contiguous across process 27ranks means that the first address in the memory segment of process i is consecutive with 28the last address in the memory segment of process i-1. This may enable the user to 29calculate remote address offsets with local information only.

If the Fortran compiler provides TYPE(C\_PTR), then the following interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR, but with a different linker name:

```
INTERFACE MPI_WIN_ALLOCATE_SHARED
                                                                                   35
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
                                                                                   36
            BASEPTR, WIN, IERROR)
                                                                                   37
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                   38
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
                                                                                   39
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
                                                                                   40
                                                                                   41
        TYPE(C_PTR) :: BASEPTR
                                                                                   42
    END SUBROUTINE
END INTERFACE
                                                                                   43
                                                                                   44
```

The linker name base of this overloaded function is MPI\_WIN\_ALLOCATE\_SHARED\_CPTR. <sup>45</sup> The implied linker names are described in Section 17.1.5 on page 607. <sup>46</sup>

The info argument can be used to specify hints similar to the info argument for 47 MPI\_WIN\_CREATE, MPI\_WIN\_ALLOC, and MPI\_ALLOC\_MEM. The additional info key 48

#### Unofficial Draft for Comment Only

1

2 3

4

5

6

9

10

11

12

30

31

32

33

alloc\_shared\_noncontig allows the library to optimize the layout of the shared memory seg ments in memory.

Advice to users. If the info key alloc\_shared\_noncontig is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (End of advice to users.)

Advice to implementors. If the user sets the info key alloc\_shared\_noncontig to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (*End of advice to implementors.*)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the unified 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

MPI\_WIN\_FLUSH). MPI does not define semantics for accessing shared memory windows
 in the separate memory model.

```
MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)
```

```
IN
                 win
                                            shared memory window object (handle)
25
26
       IN
                 rank
                                            rank in the group of window win (non-negative inte-
27
                                            ger)
28
       OUT
                 size
                                            size of the window segment (non-negative integer)
29
       OUT
                 disp_unit
                                            local unit size for displacements, in bytes (positive in-
30
                                             teger)
^{31}
32
       OUT
                 baseptr
                                            address for load/store access to window segment (choice)
33
34
     int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
35
                    int *disp_unit, void *baseptr)
36
     MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) BIND(C)
37
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
38
          TYPE(MPI_Win), INTENT(IN) :: win
39
          INTEGER, INTENT(IN) :: rank
40
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
                                                                size
41
          INTEGER, INTENT(OUT) :: disp_unit
42
          TYPE(C_PTR), INTENT(OUT) :: baseptr
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
46
          INTEGER WIN, RANK, DISP_UNIT, IERROR
47
          INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
48
```

3 4

5 6

> 7 8

9

10

11

12

13

14 15

22 23

 $^{24}$ 

This function queries the process-local address for remote memory segments created with MPI\_WIN\_ALLOCATE\_SHARED. This function can return different process-local ad-3 dresses for the same physical memory on different processes. The returned memory can be 4 used for load/store accesses subject to the constraints defined in Section 11.7. This function 5can only be called with windows of type MPI\_WIN\_FLAVOR\_SHARED. If the passed window is not of flavor MPI\_WIN\_FLAVOR\_SHARED, the error MPI\_ERR\_RMA\_WRONG\_FLAVOR is 6  $\overline{7}$ raised. When rank is MPI\_PROC\_NULL, the pointer, disp\_unit, and size returned are the pointer, disp\_unit, and size of the memory segment belonging the lowest rank that specified 9 size > 0. If all processes in the group attached to the window specified size = 0, then the call returns size = 0 and a baseptr as if MPI\_ALLOC\_MEM was called with size = 0. 10

If the Fortran compiler provides TYPE(C\_PTR), then the following interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR. but with a different linker name:

```
INTERFACE MPI_WIN_SHARED_QUERY
   SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
           BASEPTR, IERROR)
       USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
       INTEGER :: WIN, RANK, DISP_UNIT, IERROR
       INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                           SIZE
       TYPE(C_PTR) :: BASEPTR
   END SUBROUTINE
END INTERFACE
```

The linker name base of this overloaded function is MPI\_WIN\_SHARED\_QUERY\_CPTR. The implied linker names are described in Section 17.1.5 on page 607.

#### 11.2.4 Window of Dynamically Attached Memory

The MPI-2 RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make onesided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using malloc or new respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory. In addition, there is no easy way to handle the situation where the predefined amount of memory turns out to be inadequate. To support this model, the routine MPI\_WIN\_CREATE\_DYNAMIC creates a window that makes it possible to expose memory without remote synchronization. It must be used in combination with the local routines MPI\_WIN\_ATTACH and MPI\_WIN\_DETACH.

> 4546

48

1

 $\mathbf{2}$ 

8

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

43

1 MPI\_WIN\_CREATE\_DYNAMIC(info, comm, win) 2 IN info info argument (handle) 3 IN intra-communicator (handle) comm 4 5OUT window object returned by the call (handle) win 6  $\overline{7}$ int MPI\_Win\_create\_dynamic(MPI\_Info info, MPI\_Comm comm, MPI\_Win \*win) 8 MPI\_Win\_create\_dynamic(info, comm, win, ierror) BIND(C) 9 TYPE(MPI\_Info), INTENT(IN) :: info 10 TYPE(MPI\_Comm), INTENT(IN) :: comm 11 TYPE(MPI\_Win), INTENT(OUT) :: win 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14MPI\_WIN\_CREATE\_DYNAMIC(INFO, COMM, WIN, IERROR) 15INTEGER INFO, COMM, WIN, IERROR 1617This is a collective call executed by all processes in the group of comm. It returns a window win without memory attached. Existing process memory can be attached as 18 described below. This routine returns a window object that can be used by these processes to 19perform RMA operations on attached memory. Because this window has special properties, 20it will sometimes be referred to as a *dynamic* window. 21The info argument can be used to specify hints similar to the info argument for 22MPI\_WIN\_CREATE. 23In the case of a window created with MPI\_WIN\_CREATE\_DYNAMIC, the target\_disp for  $^{24}$ all RMA functions is the address at the target; i.e., the effective window\_base is MPI\_BOTTOM 2526and the disp\_unit is one. 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 27origin process. 2829 Advice to implementors. In environments with heterogeneous data representations, 30 care must be exercised in communicating addresses between processes. For example, 31it is possible that an address valid at the target process (for example, a 64-bit pointer) 32 cannot be expressed as an address at the origin (for example, the origin uses 32-bit 33 pointers). For this reason, a portable MPI implementation should ensure that the type 34 MPI\_AINT (see Table 3.3 on Page 27) is able to store addresses from any process. (End 35of advice to implementors.) 36 37 Memory in this window may not be used as the target of one-sided accesses in this 38 window until it is attached using the function MPI\_WIN\_ATTACH. That is, in addition to 39 using MPI\_WIN\_CREATE\_DYNAMIC to create an MPI window, the user must use 40 MPI\_WIN\_ATTACH before any local memory may be the target of an MPI RMA operation. 41 Only memory that is currently accessible may be attached. 4243 44454647 48

#### MPI\_WIN\_ATTACH(win, base, size) IN window object (handle) win IN base initial address of memory to be attached IN size size of memory to be attached in bytes int MPI\_Win\_attach(MPI\_Win win, void \*base, MPI\_Aint size) MPI\_Win\_attach(win, base, size, ierror) BIND(C) TYPE(MPI\_Win), INTENT(IN) :: win TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: base INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_WIN\_ATTACH(WIN, BASE, SIZE, IERROR) INTEGER WIN, IERROR <type> BASE(\*) INTEGER (KIND=MPI\_ADDRESS\_KIND) SIZE

Attaches a local memory region beginning at **base** for remote access within the given window. The memory region specified must not contain any part that is already attached to the window win, that is, attaching overlapping memory concurrently within the same window is erroneous. The argument win must be a window that was created with MPI\_WIN\_CREATE\_DYNAMIC. Multiple (but non-overlapping) memory regions may be attached to the same window.

*Rationale.* Requiring that memory be explicitly attached before it is exposed to one-sided access by other processes can significantly simplify implementations and improve performance. The ability to make memory available for RMA operations without requiring a collective MPI\_WIN\_CREATE call is needed for some one-sided programming models. (*End of rationale.*)

Advice to users. Attaching memory to a window may require the use of scarce resources; thus, attaching large regions of memory is not recommended in portable programs. Attaching memory to a window may fail if sufficient resources are not available; this is similar to the behavior of MPI\_ALLOC\_MEM.

The user is also responsible for ensuring that MPI\_WIN\_ATTACH at the target has returned before a process attempts to target that memory with an MPI RMA call.

Performing an RMA operation to memory that has not been attached to a window created with MPI\_WIN\_CREATE\_DYNAMIC is erroneous. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to make as much memory available for attaching as possible. Any limitations should be documented by the implementor. (*End of advice to implementors.*)

Attaching memory is a local operation as defined by MPI, which means that the call <sup>45</sup> is not collective and completes without requiring any MPI routine to be called in any other <sup>46</sup> process. Memory may be detached with the routine MPI\_WIN\_DETACH. After memory has <sup>47</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 41

42

43

44

```
1
     been detached, it may not be the target of an MPI RMA operation on that window (unless
\mathbf{2}
     the memory is re-attached with MPI_WIN_ATTACH).
3
4
     MPI_WIN_DETACH(win, base)
5
6
       IN
                                             window object (handle)
                 win
7
       IN
                 base
                                             initial address of memory to be detached
8
9
     int MPI_Win_detach(MPI_Win win, const void *base)
10
11
     MPI_Win_detach(win, base, ierror) BIND(C)
12
          TYPE(MPI_Win), INTENT(IN) :: win
13
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_WIN_DETACH(WIN, BASE, IERROR)
16
          INTEGER WIN, IERROR
17
          <type> BASE(*)
18
19
         Detaches a previously attached memory region beginning at base. The arguments base
20
     and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.
21
22
           Advice to users. Detaching memory may permit the implementation to make more
23
           efficient use of special memory or provide memory that may be needed by a subsequent
^{24}
           MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed.
25
           Memory should be detached before it is freed by the user. (End of advice to users.)
26
27
          Memory becomes detached when the associated dynamic memory window is freed, see
     Section 11.2.5.
28
29
30
     11.2.5 Window Destruction
^{31}
32
33
     MPI_WIN_FREE(win)
34
       INOUT
                 win
                                             window object (handle)
35
36
     int MPI_Win_free(MPI_Win *win)
37
38
     MPI_Win_free(win, ierror) BIND(C)
39
          TYPE(MPI_Win), INTENT(INOUT) :: win
40
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_WIN_FREE(WIN, IERROR)
          INTEGER WIN, IERROR
43
44
         Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This
45
     is a collective call executed by all processes in the group associated with
46
     win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its
47
```

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 3 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 4 the window was created with MPI\_WIN\_ALLOCATE, MPI\_WIN\_FREE will free the window memory that was allocated in MPI\_WIN\_ALLOCATE. 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 called free. This is 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.)

#### Window Attributes 11.2.6

The following attributes are cached with a window when the window is created.

		21
MPI_WIN_BASE	window base address.	22
MPI_WIN_SIZE	window size, in bytes.	20
MPI_WIN_DISP_UNIT	displacement unit associated with the window.	23
MPI_WIN_CREATE_FLAVOR	how the window was created.	24
MPI_WIN_MODEL	memory model for window.	25
	memory model for window.	26

In C, calls to MPI\_Win\_get\_attr(win, MPI\_WIN\_BASE, &base, &flag), MPI\_Win\_get\_attr(win, MPI\_WIN\_SIZE, &size, &flag), MPI\_Win\_get\_attr(win, MPI\_WIN\_DISP\_UNIT, &disp\_unit, &flag), MPI\_Win\_get\_attr(win, MPI\_WIN\_CREATE\_FLAVOR, &create\_kind, &flag), and MPI\_Win\_get\_attr(win, MPI\_WIN\_MODEL, &memory\_model, &flag) will return in base a pointer to the start of the window win, and will return in size, disp\_unit, create\_kind, and memory\_model pointers to the size, displacement unit of the window, the kind of routine used to create the window, and the memory model, respectively. A detailed listing of the type of the pointer in the attribute value argument to MPI\_WIN\_GET\_ATTR and MPI\_WIN\_SET\_ATTR is shown in Table 11.1.

Attribute	C Type
MPI_WIN_BASE	void *
MPI_WIN_SIZE	MPI_Aint *
MPI_WIN_DISP_UNIT	int *
MPI_WIN_CREATE_FLAVOR	int *
MPI_WIN_MODEL	int *

Table 11.1: C types of attribute value argument to MPI\_WIN\_GET\_ATTR and MPI\_WIN\_SET\_ATTR.

#### **Unofficial Draft for Comment Only**

1 2

5

6  $\overline{7}$ 

8 9

10

11

12

13

14

15

1617 18

19

20

27

28

29

30

31

32

33

34

35

46

1		,		_ATTR(win, MPI_WIN_BASE, base, flag, ierror),	
2	MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror),				
3		•		DISP_UNIT, disp_unit, flag, ierror),	
4		•		$CREATE_FLAVOR$ , create_kind, flag, ierror), and	
5		(		MODEL, memory_model, flag, ierror) will return in	
6	base, size	, disp_unit create_	kind and men	nory_model the (integer representation of) the base	
7	address,	the size, the disp	olacement uni	t of the window win, the kind of routine used to	
8	create the	e window, and th	e memory mo	del, respectively.	
9	The	values of create_	kind are		
10					
11		_FLAVOR_CREATE		Window was created with MPI_WIN_CREATE.	
12		_FLAVOR_ALLOCA		Window was created with MPI_WIN_ALLOCATE.	
13	MPI_WIN	_FLAVOR_DYNAM	IC	Window was created with	
14				MPI_WIN_CREATE_DYNAMIC.	
15	MPI_WIN	_FLAVOR_SHARED	)	Window was created with	
16				MPI_WIN_ALLOCATE_SHARED.	
17	The	values of memory_	model are MP	I_WIN_SEPARATE and MPI_WIN_UNIFIED. The mean-	
18	ing of the	ese is described in	Section 11.4.		
19	In th	ne case of window	s created with	h MPI_WIN_CREATE_DYNAMIC, the base address	
20	is MPI_BC	OTTOM and the si	ze is 0. In C,	pointers are returned and in Fortran, the values are	
21	returned,	for the respective	e attributes. (	(The window attribute access functions are defined	
22	in Section	n 6.7.3, page 272.	.) The value :	returned for an attribute on a window is constant	
23	over the l	lifetime of the wir	ndow.		
24	The	other "window at	tribute," nam	hely the group of processes attached to the window,	
25	can be re	trieved using the	call below.		
26					
27		I_GET_GROUP(w	(in group)		
28 29			m, group)		
30	IN	win		window object (handle)	
31	OUT	group		group of processes which share access to the window	
32				(handle)	
33					
34	int MPI_	Win_get_group(	MPI_Win win	, MPI_Group *group)	
35					
36		_get_group(win,			
37		E(MPI_Win), INT			
38		E(MPI_Group), I		•	
39	TNLF	EGER, OPTIONAL,	INTENT (OUT)	) :: ierror	
40	MPI_WIN_	GET_GROUP(WIN,	GROUP, IER	ROR)	
41		EGER WIN, GROUP			
42					
43				duplicate of the group of the communicator used to	
44	create the	e window associat	ted with win.	The group is returned in group.	
45					
46	11.2.7	Window Info			

<sup>47</sup> Hints specified via info (see Section 9, page 367) allow a user to provide information to
 <sup>48</sup> direct optimization. Providing hints may enable an implementation to deliver increased

#### Unofficial Draft for Comment Only

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 this hint. (*End of advice to implementors.*)

MPI\_WIN\_SET\_INFO(win, info)

INOUT	win	window object (handle)
IN	info	info object (handle)

int MPI\_Win\_set\_info(MPI\_Win win, MPI\_Info info)

```
MPI_WIN_SET_INFO(WIN, INFO, IERROR)
INTEGER WIN, INFO, IERROR
```

MPI\_WIN\_SET\_INFO sets new values for the hints of the window associated with win. The call is collective on the group of win. 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 window cannot easily be changed once the window 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\_WIN\_GET\_INFO(win, info\_used)

IN	win	window object (handle)
OUT	info_used	new info object (handle)
int MPI_	Win_get_info(MPI_Win win,	MPI_Info *info_used)

MPI\_WIN\_GET\_INFO(WIN, INFO\_USED, IERROR) INTEGER WIN, INFO\_USED, IERROR

MPI\_WIN\_GET\_INFO returns a new info object containing the hints of the window 46 associated with win. The current setting of all hints actually used by the system related to 47 this window is returned in info\_used. If no such hints exist, a handle to a newly created 48

#### Unofficial Draft for Comment Only

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 window. This set of hints may be greater or smaller than the set of hints specified when the window 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.)

# 11.3 Communication Calls

<sup>12</sup> MPI supports the following RMA communication calls: MPI\_PUT and MPI\_RPUT trans-<sup>14</sup> fer data from the caller memory (origin) to the target memory; MPI\_GET and MPI\_RGET <sup>15</sup> transfer data from the target memory to the caller memory; MPI\_ACCUMULATE and <sup>16</sup> MPI\_RACCUMULATE update locations in the target memory, e.g., by adding to these lo-<sup>17</sup> cations values sent from the caller memory; MPI\_GET\_ACCUMULATE,

17MPI\_RGET\_ACCUMULATE and MPI\_FETCH\_AND\_OP perform atomic read-modify-write 18 and return the data before the accumulate operation; and MPI\_COMPARE\_AND\_SWAP 19 performs a remote atomic compare and swap operation. These operations are *nonblocking*: 20the call initiates the transfer, but the transfer may continue after the call returns. The 21transfer is completed, at the origin or both the origin and the target, when a subsequent 22 synchronization call is issued by the caller on the involved window object. These synchro-23nization calls are described in Section 11.5, page 437. Transfers can also be completed 24with calls to flush routines; see Section 11.5.4, page 449 for details. For the MPI\_RPUT, 25MPI\_RGET, MPI\_RACCUMULATE, and MPI\_RGET\_ACCUMULATE calls, the transfer can 26be locally completed by using the MPI test or wait operations described in Section 3.7.3. 27page 52. 28

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 un-32 defined; if a location is updated by a put or accumulate operation, then the outcome of 33 loads or other RMA operations is undefined until the updating operation has completed at 34 the target. There is one exception to this rule; namely, the same location can be updated 35 by several concurrent accumulate calls, the outcome being as if these updates occurred 36 in some order. In addition, the outcome of concurrent load/store and RMA updates to 37 the same memory location is undefined. These restrictions are described in more detail in 38 Section 11.7, page 453. 39

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 message passing. It simplifies some coding, and is very useful with accumulate operations, to
 allow atomic updates of local variables. (*End of rationale.*)

### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8 9 10

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.

MPI_PUT(origin_addr, origin_count, origin_datatype,	target_rank, target_disp, target_count,
target_datatype, win)	

IN	origin_addr	initial address of origin buffer (choice)	15
IN	origin_count	number of entries in origin buffer (non-negative inte-	16
		ger)	17
IN	origin_datatype	datatype of each entry in origin buffer (handle)	18
IN	target_rank	rank of target (non-negative integer)	19
IIN		Tank of target (non-negative integer)	20
IN	target_disp	displacement from start of window to target buffer	21
		(non-negative integer)	22
IN	target_count	number of entries in target buffer (non-negative inte-	23
		ger)	24
		- ,	25
IN	target_datatype	datatype of each entry in target buffer (handle)	26
IN	win	window object used for communication (handle)	27

int MPI\_Put(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\_Put(origin\_addr, origin\_count, origin\_datatype, target\_rank,

target\_disp, target\_count, target\_datatype, win, ierror) BIND(C) TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI\_PUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, IERROR) <type> ORIGIN\_ADDR(\*) INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, WIN, IERROR

1

2

3

4 56

7

8

9

2829

30

31

32

33

34

35

36

37

38

39

40

41 42

43

44

45

46

47

<sup>1</sup> Transfers origin\_count successive entries of the type specified by the origin\_datatype, <sup>2</sup> starting at address origin\_addr on the origin node to the target node specified by the <sup>3</sup> win, target\_rank pair. The data are written in the target buffer at address target\_addr = <sup>4</sup> window\_base + target\_disp×disp\_unit, where window\_base and disp\_unit are the base address <sup>5</sup> and window displacement unit specified at window initialization, by the target process. <sup>6</sup> The target buffer is specified by the arguments target\_count and target\_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin\_addr, origin\_count, origin\_datatype, target\_rank, tag, comm, and the target process executed a receive operation with arguments target\_addr, target\_count, target\_datatype, source, tag, comm, where target\_addr is the target buffer address computed as explained above, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.

The communication must satisfy the same constraints as for a similar message-passing communication. The target\_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window or in attached memory in a dynamic window.

<sup>17</sup> The target\_datatype argument is a handle to a datatype object defined at the origin <sup>18</sup> process. However, this object is interpreted at the target process: the outcome is as if <sup>19</sup> the target datatype object was defined at the target process by the same sequence of calls <sup>20</sup> used to define it at the origin process. The target datatype must contain only relative <sup>21</sup> displacements, not absolute addresses. The same holds for get and accumulate.

22 23

 $^{24}$ 

25 26

27

Advice to users. The target\_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment if only portable datatypes are used (portable datatypes are defined in Section 2.4, page 11).

The performance of a put transfer can be significantly affected, on some systems, 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.*)

35 Advice to implementors. A high-quality implementation will attempt to prevent 36 remote accesses to memory outside the window that was exposed by the process. 37 This, both for debugging purposes, and for protection with client-server codes that 38 use RMA. I.e., a high-quality implementation will check, if possible, window bounds 39 on each RMA call, and raise an MPI exception at the origin call if an out-of-bound 40 situation occurred. Note that the condition can be checked at the origin. Of course, 41 the added safety achieved by such checks has to be weighed against the added cost of 42such checks. (End of advice to implementors.)

- 43
- $44 \\ 45$
- 46
- 47
- 48

Similar to MPI\_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin\_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window or within attached memory in a dynamic window, and the copied data must fit, without truncation, in the origin buffer.

43

44

45

```
1
     11.3.3 Examples for Communication Calls
\mathbf{2}
     These examples show the use of the MPI_GET function. As all MPI RMA communication
3
     functions are nonblocking, they must be completed. In the following, this is accomplished
4
     with the routine MPI_WIN_FENCE, introduced in Section 11.5.
5
6
     Example 11.1 We show how to implement the generic indirect assignment A = B(map),
7
     where A, B and map have the same distribution, and map is a permutation. To simplify, we
8
     assume a block distribution with equal size blocks.
9
10
    SUBROUTINE MAPVALS(A, B, map, m, comm, p)
11
     USE MPI
12
     INTEGER m, map(m), comm, p
13
    REAL A(m), B(m)
14
15
     INTEGER otype(p), oindex(m),
                                     & ! used to construct origin datatypes
16
          17
          count(p), total(p),
                                     &
18
          disp_int, win, ierr
19
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
20
21
     ! This part does the work that depends on the locations of B.
22
     ! Can be reused while this does not change
23
^{24}
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
25
     disp_int = realextent
26
     size = m * realextent
27
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                                &
28
                           comm, win, ierr)
29
30
     ! This part does the work that depends on the value of map and
31
     ! the locations of the arrays.
32
     ! Can be reused while these do not change
33
34
     ! Compute number of entries to be received from each process
35
36
    DO i=1,p
37
       count(i) = 0
38
     END DO
39
    DO i=1,m
40
       j = map(i)/m+1
41
       count(j) = count(j)+1
42
    END DO
43
44
    total(1) = 0
45
    DO i=2,p
46
       total(i) = total(i-1) + count(i-1)
47
     END DO
48
```

```
1
DO i=1,p
                                                                                      \mathbf{2}
  count(i) = 0
                                                                                      3
END DO
                                                                                      4
! compute origin and target indices of entries.
                                                                                      5
                                                                                      6
! entry i at current process is received from location
                                                                                      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
  k = MOD(map(i), m) + 1
                                                                                      12
  count(j) = count(j)+1
                                                                                      13
  oindex(total(j) + count(j)) = i
                                                                                      14
                                                                                      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, &
                                        oindex(total(i)+1:total(i)+count(i)), &
                                                                                      21
                                        MPI_REAL, otype(i), ierr)
                                                                                      22
                                                                                      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)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
                                                                                      27
END DO
                                                                                      28
                                                                                      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)
END DO
                                                                                      35
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                      36
                                                                                      37
                                                                                      38
CALL MPI_WIN_FREE(win, ierr)
                                                                                      39
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
                                                                                      40
                                                                                      41
  CALL MPI_TYPE_FREE(ttype(i), ierr)
                                                                                      42
END DO
RETURN
                                                                                      43
                                                                                      44
END
                                                                                      45
                                                                                      46
Example 11.2
                                                                                      47
                                                                                      48
```

&

1 A simpler version can be written that does not require that a datatype be built for the  $\mathbf{2}$ target buffer. But, one then needs a separate get call for each entry, as illustrated below. 3 This code is much simpler, but usually much less efficient, for large arrays. 4 SUBROUTINE MAPVALS(A, B, map, m, comm, p) 56 USE MPI 7INTEGER m, map(m), comm, p REAL A(m), B(m)8 9 INTEGER disp\_int, win, ierr 10 INTEGER (KIND=MPI\_ADDRESS\_KIND) lowerbound, size, realextent, disp\_aint 11CALL MPI\_TYPE\_GET\_EXTENT(MPI\_REAL, lowerbound, realextent, ierr) 12disp\_int = realextent 13

```
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
```

```
<sup>16</sup> comm, win, ierr)
```

18 CALL MPI\_WIN\_FENCE(0, win, ierr) 19DO i=1,m j = map(i)/m20disp\_aint = MOD(map(i),m) 21CALL MPI\_GET(A(i), 1, MPI\_REAL, j, disp\_aint, 1, MPI\_REAL, win, ierr) 22 23END DO  $^{24}$ CALL MPI\_WIN\_FENCE(0, win, ierr) CALL MPI\_WIN\_FREE(win, ierr) 2526RETURN END

27 28

17

<sup>29</sup> 11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. This will allow, for example, the accumulation of a sum by having all involved processes add their contribution to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 11.7 for details.

38 Accumulate Function

39

36 37

- 40
- $41 \\ 42$

- 44
- 45
- 46
- 47
- 48

MP	MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)		
IN	origin_addr	initial address of buffer (choice)	3 4
IN	origin_count	number of entries in buffer (non-negative integer)	4 5
IN	origin_datatype	datatype of each entry (handle)	6
IN	target_rank	rank of target (non-negative integer)	7
IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	8 9 10
IN	target_count	number of entries in target buffer (non-negative integer)	11 12
IN	target_datatype	datatype of each entry in target buffer (handle)	13 14
IN	ор	reduce operation (handle)	14 15
IN	win	window object (handle)	16
			17
int	MPI_Accumulate(const void *ori		18 19
MPI_Datatype origin_datatype, int target_rank,			20
	MPI_Aint target_disp	-	21
	MPI_Datatype target_0	datatype, MPI_Op op, MPI_Win win)	22
MPI	_Accumulate(origin_addr, origir	n_count, origin_datatype, target_rank,	23
		count, target_datatype, op, win, ierror)	24
	BIND(C)	- /\	25
		C(IN), ASYNCHRONOUS :: origin_addr	26
	-	n_count, target_rank, target_count	27
		:: origin_datatype, target_datatype	28
	<pre>INTEGER(KIND=MPI_ADDRESS_KIND) TYPE(MPI_Op), INTENT(IN) :: content</pre>	pp	29
	TYPE(MPI_Up), INTENT(IN) ::	-	30 31
	INTEGER, OPTIONAL, INTENT(OUT)		31
			33
MPI		N_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	34
	-	COUNT, TARGET_DATATYPE, OP, WIN, IERROR)	35
	<pre><type> ORIGIN_ADDR(*) INTEGED(KIND_MDI_ADDREGG_KIND)</type></pre>		36
	INTEGER (KIND=MPI_ADDRESS_KIND)	) TARGET_DISP DATATYPE,TARGET_RANK, TARGET_COUNT,	37
	TARGET_DATATYPE, OP, WIN, IERF		38
			39
	Accumulate the contents of the origin	n buffer (as defined by origin_addr, origin_count and	40

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.

 $^{41}$ 

#### Unofficial Draft for Comment Only

```
1
          Each datatype argument must be a predefined datatype or a derived datatype, where
\mathbf{2}
     all basic components are of the same predefined datatype. Both datatype arguments must
3
     be constructed from the same predefined datatype. The operation op applies to elements of
4
     that predefined type. The parameter target_datatype must not specify overlapping entries,
\mathbf{5}
     and the target buffer must fit in the target window.
6
          A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative
7
     function f(a,b) = b; i.e., the current value in the target memory is replaced by the value
8
     supplied by the origin.
9
          MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE,
10
     MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not
11
     in collective reduction operations such as MPI_REDUCE.
12
                             MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
           Advice to users.
13
           eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
14
           different constraints on concurrent updates. (End of advice to users.)
15
16
17
     Example 11.3 We want to compute B(j) = \sum_{map(i)=j} A(i). The arrays A, B and map are
18
     distributed in the same manner. We write the simple version.
19
20
     SUBROUTINE SUM(A, B, map, m, comm, p)
21
     USE MPI
22
     INTEGER m, map(m), comm, p, win, ierr, disp_int
23
     REAL A(m), B(m)
^{24}
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
25
26
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
27
     size = m * realextent
28
     disp_int = realextent
29
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
30
                            comm, win, ierr)
^{31}
32
     CALL MPI_WIN_FENCE(0, win, ierr)
33
     DO i=1,m
34
        j = map(i)/m
35
       disp_aint = MOD(map(i),m)
36
       CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
                                                                                     &
37
                              MPI_SUM, win, ierr)
38
     END DO
39
     CALL MPI_WIN_FENCE(0, win, ierr)
40
^{41}
     CALL MPI_WIN_FREE(win, ierr)
42
     RETURN
43
     END
44
45
          This code is identical to the code in Example 11.2, page 423, except that a call to
```

get has been replaced by a call to accumulate. (Note that, if map is one-to-one, then the code computes  $B = A(map^{-1})$ , which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 11.1, page 422,

#### Unofficial Draft for Comment Only

the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

#### Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 11.7 for details). The predefined operation MPI\_REPLACE provides fetch-and-set behavior.

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)

		,		16
I	N (	origin_addr	initial address of buffer (choice)	17
I	N d	origin_count	number of entries in origin buffer (non-negative inte-	18
			ger)	19
I	N d	origin_datatype	datatype of each entry in origin buffer (handle)	20 21
C	DUT	result_addr	initial address of result buffer (choice)	22
I	N	result_count	number of entries in result buffer (non-negative inte- ger)	23 24
I	N	result_datatype	datatype of each entry in result buffer (handle)	25 26
I	N t	target_rank	rank of target (non-negative integer)	27
I	N	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	28 29
I	N	target_count	number of entries in target buffer (non-negative inte- ger)	30 31 32
I	N t	target_datatype	datatype of each entry in target buffer (handle)	33
I	N	ор	reduce operation (handle)	34
		win	window object (handle)	35
		vviii	window object (nandie)	36
int	- MPT Got	t accumulate(const void >	*origin_addr, int origin_count,	37
1110			atatype, void *result_addr,	38
			_Datatype result_datatype,	39 40
		int target_rank, MPI_	Aint target_disp, int target_count,	40
		MPI_Datatype target_d	atatype, MPI_Op op, MPI_Win win)	42
MPI	[_Get_aco	cumulate(origin_addr, or:	igin_count, origin_datatype, result_addr,	43
		0	datatype, target_rank, target_disp,	44
			datatype, op, win, ierror) BIND(C)	45
			(IN), ASYNCHRONOUS :: origin_addr	46
	TYPE(*)	), DIMENSION(), ASYNCH	RONOUS :: result_addr	47

 $\frac{3}{4}$ 

 $13 \\ 14$ 

1 INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank,  $\mathbf{2}$ target\_count 3 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype, 4 result\_datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 56 TYPE(MPI\_Op), INTENT(IN) :: op 7 TYPE(MPI\_Win), INTENT(IN) :: win 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 MPI\_GET\_ACCUMULATE(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_ADDR, 10 RESULT\_COUNT, RESULT\_DATATYPE, TARGET\_RANK, TARGET\_DISP, 11 TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, IERROR) 12<type> ORIGIN\_ADDR(\*), RESULT\_ADDR(\*) 13 INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 14INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_COUNT, RESULT\_DATATYPE, 15TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, IERROR 1617 Accumulate origin\_count elements of type origin\_datatype from the origin buffer ( 18 origin\_addr) to the buffer at offset target\_disp, in the target window specified by target\_rank 19and win, using the operation op and return in the result buffer result\_addr the content of 20the target buffer before the accumulation. 21The origin and result buffers (origin\_addr and result\_addr) must be disjoint. Each 22 datatype argument must be a predefined datatype or a derived datatype where all basic 23components are of the same predefined datatype. All datatype arguments must be con-24structed from the same predefined datatype. The operation op applies to elements of that 25predefined type. target\_datatype must not specify overlapping entries, and the target buffer 26must fit in the target window or in attached memory in a dynamic window. The operation 27is executed atomically for each basic datatype; see Section 11.7 for details. 28Any of the predefined operations for MPI\_REDUCE, and MPI\_NO\_OP or MPI\_REPLACE 29 can be specified as op. User-defined functions cannot be used. A new predefined operation, 30 MPI\_NO\_OP, is defined. It corresponds to the associative function f(a,b) = a; i.e., the  $^{31}$ current value in the target memory is returned in the result buffer at the origin and no 32 operation is performed on the target buffer. MPI\_NO\_OP can be used only in 33 MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP. 34MPI\_NO\_OP cannot be used in MPI\_ACCUMULATE, MPI\_RACCUMULATE, or collective 35 reduction operations, such as MPI\_REDUCE and others. 36 37 Advice to users. MPI\_GET is similar to MPI\_GET\_ACCUMULATE, with the opera-38 tion MPI\_NO\_OP. Note, however, that MPI\_GET and MPI\_GET\_ACCUMULATE have different constraints on concurrent updates. (End of advice to users.) 39 40  $^{41}$ Fetch and Op Function 42The generic functionality of MPI\_GET\_ACCUMULATE might limit the performance of fetch-43 and-increment or fetch-and-add calls that might be supported by special hardware oper-44 ations. MPI\_FETCH\_AND\_OP thus allows for a fast implementation of a commonly used 45 subset of the functionality of MPI\_GET\_ACCUMULATE. 46 4748

MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win) <sup>1</sup>			
			2
IN	origin_addr	initial address of buffer (choice)	$\frac{3}{4}$
0	JT result_addr	initial address of result buffer (choice)	4 5
IN	datatype	datatype of the entry in origin, result, and target buf-	6
		fers (handle)	7
IN	target_rank	rank of target (non-negative integer)	8 9
IN	target_disp	displacement from start of window to beginning of tar-	10
	0	get buffer (non-negative integer)	11
IN	ор	reduce operation (handle)	12
IN	win	window object (handle)	13
			14
int	MPI_Fetch_and_op(const void *c	origin_addr, void *result_addr,	15 16
	_	e, int target_rank, MPI_Aint target_disp,	10
	MPI_Op op, MPI_Win win)		
мрт	MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,		
	<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: origin_addr</pre>		
	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
	INTEGER, INTENT(IN) :: target_rank		
	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp <sup>21</sup>		
	TYPE(MPI_Op), INTENT(IN) :: op		
	TYPE(MPI_Win), INTENT(IN) :: win		
	INTEGER, OPTIONAL, INTENT(OUT)	) :: ierror	28 29
MPI.	_FETCH_AND_OP(ORIGIN_ADDR, RESU	JLT_ADDR, DATATYPE, TARGET_RANK,	30
	TARGET_DISP, OP, WIN, IERROR)		
	<type> ORIGIN_ADDR(*), RESULT_</type>	_ADDR(*)	32
	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP		
	INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR		
	Accumulate one element of type datatype from the origin buffer (origin_addr) to the		

Accumulate one element of type 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. 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. The datatype argument must be a predefined datatype. The operation is executed atomically.

#### Compare and Swap Function

Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at origin and target are equal.

#### Unofficial Draft for Comment Only

1 2	MPI_COM	PARE_AND_SWAP(or target_disp, win)	igin_addr, compare_addr, result_addr, datatype, target_rank,
3	IN	origin_addr	initial address of buffer (choice)
4 5	IN	compare_addr	initial address of compare buffer (choice)
6	OUT	result_addr	initial address of result buffer (choice)
7	IN	datatype	datatype of the element in all buffers (handle)
8 9	IN	target_rank	rank of target (non-negative integer)
10 11	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)
12 13	IN	win	window object (handle)
14 15 16 17	int MPI_(	void *result_a	nst void *origin_addr, const void *compare_addr, addr, MPI_Datatype datatype, int target_rank, et_disp, MPI_Win win)
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33	<pre>MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,</pre>		
34 35 36 37 38 39 40 41 42	This function compares one element of type datatype in the compare buffer 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.		
43 44	11.3.5 R	equest-based RMA Co	ommunication Operations
45 46 47 48	with the $F$ functions of	RMA operations and to	tion operations allow the user to associate a request handle est or wait for the completion of these requests using the 7.3, page 52. Request-based RMA operations are only valid

# Unofficial Draft for Comment Only

Upon returning from a completion call in which an RMA operation completes, the MPI\_ERROR field in the associated status object is set appropriately (see Section 3.2.5 on page 30). All other fields of status and the results of status query functions (e.g., MPI\_GET\_COUNT) are undefined. It is valid to mix different request types (e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI\_WAITALL). It is erroneous to call MPI\_REQUEST\_FREE or MPI\_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The end of the epoch, or explicit bulk synchronization using MPI\_WIN\_FLUSH, MPI\_WIN\_FLUSH\_ALL, MPI\_WIN\_FLUSH\_LOCAL or MPI\_WIN\_FLUSH\_LOCAL\_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally.

MPI\_RPUT(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, win, request)

IN	origin_addr	initial address of origin buffer (choice)	18
IN	origin_count	number of entries in origin buffer (non-negative integer)	19 20
IN	origin_datatype	datatype of each entry in origin buffer (handle)	21 22
IN	target_rank	rank of target (non-negative integer)	23
IN	target_disp	displacement from start of window to target buffer (non-negative integer)	24 25
IN	target_count	number of entries in target buffer (non-negative inte- ger)	26 27 28
IN	target_datatype	datatype of each entry in target buffer (handle)	29
IN	win	window object used for communication (handle)	30 31
OUT	request	RMA request (handle)	31

<pre>int MPI_Rput(const void *origin_addr, int origin_count,</pre>	34
MPI_Datatype origin_datatype, int target_rank,	35
MPI_Aint target_disp, int target_count,	36
MPI_Datatype target_datatype, MPI_Win win,	37
MPI_Request *request)	38
	39

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
3
                     TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
4
                     IERROR)
5
          <type> ORIGIN_ADDR(*)
6
          INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
7
          INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
8
          TARGET_DATATYPE, WIN, REQUEST, IERROR
9
10
          MPI_RPUT is similar to MPI_PUT (Section 11.3.1), except that it allocates a commu-
11
     nication request object and associates it with the request handle (the argument request).
12
     The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) in-
13
     dicates that the sender is now free to update the locations in the origin buffer. It does
14
     not indicate that the data is available at the target window. If remote completion is re-
15
     quired, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK or
16
     MPI_WIN_UNLOCK_ALL can be used.
17
18
     MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count,
19
                     target_datatype, win, 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-
23
                                              ger)
^{24}
       IN
                 origin_datatype
                                              datatype of each entry in origin buffer (handle)
25
26
       IN
                 target_rank
                                              rank of target (non-negative integer)
27
       IN
                 target_disp
                                              displacement from window start to the beginning of
28
                                              the target buffer (non-negative integer)
29
       IN
                                              number of entries in target buffer (non-negative inte-
                 target_count
30
                                              ger)
^{31}
32
       IN
                 target_datatype
                                              datatype of each entry in target buffer (handle)
33
       IN
                                              window object used for communication (handle)
                 win
34
       OUT
                 request
                                              RMA request (handle)
35
36
37
     int MPI_Rget(void *origin_addr, int origin_count,
38
                     MPI_Datatype origin_datatype, int target_rank,
39
                     MPI_Aint target_disp, int target_count,
                     MPI_Datatype target_datatype, MPI_Win win,
40
41
                     MPI_Request *request)
42
     MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
43
                     target_disp, target_count, target_datatype, win, request,
44
                     ierror) BIND(C)
45
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
46
          INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
47
          TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
48
```

CHAPTER 11. ONE-SIDED COMMUNICATIONS

1 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp  $\mathbf{2}$ TYPE(MPI\_Win), INTENT(IN) :: win 3 TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: 4 ierror 5 MPI\_RGET(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 6 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, REQUEST, 7 IERROR) <type> ORIGIN\_ADDR(\*) 9 INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 10 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 11 TARGET\_DATATYPE, WIN, REQUEST, IERROR 1213 MPI\_RGET is similar to MPI\_GET (Section 11.3.2), except that it allocates a commu-14nication request object and associates it with the request handle (the argument request) 15that can be used to wait or test for completion. The completion of an MPI\_RGET operation 16indicates that the data is available in the origin buffer. If origin\_addr points to memory 17attached to a window, then the data becomes available in the private copy of this window. 18 19 MPI\_RACCUMULATE(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, 20target\_count, target\_datatype, op, win, request) 2122 IN origin\_addr initial address of buffer (choice) 23origin\_count number of entries in buffer (non-negative integer) IN 24origin\_datatype IN datatype of each entry in origin buffer (handle) 2526IN target\_rank rank of target (non-negative integer) 27IN target\_disp displacement from start of window to beginning of tar-28get buffer (non-negative integer) 29 IN number of entries in target buffer (non-negative intetarget\_count 30 ger) 3132 IN target\_datatype datatype of each entry in target buffer (handle) 33 IN reduce operation (handle) op 34 window object (handle) IN win 35 36 OUT request RMA request (handle) 37 38 int MPI\_Raccumulate(const void \*origin\_addr, int origin\_count, 39 MPI\_Datatype origin\_datatype, int target\_rank, 40 MPI\_Aint target\_disp, int target\_count, 41 MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win, 42MPI\_Request \*request) 43 MPI\_Raccumulate(origin\_addr, origin\_count, origin\_datatype, target\_rank, 44target\_disp, target\_count, target\_datatype, op, win, request, 45ierror) BIND(C) 46TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 47INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 48

1 2 3 4 5 6 7 8 9	<pre>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)</pre>			
10 11 12 13	<type> ORIGIN_ADDR(*) INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,</type>			
14 15 16 17 18 19 20	<sup>15</sup> MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that <sup>16</sup> it allocates a communication request object and associates it with the request handle (the <sup>17</sup> argument request) that can be used to wait or test for completion. The completion of an <sup>18</sup> MPI_RACCUMULATE operation indicates that the origin buffer is free to be updated. It <sup>19</sup> does not indicate that the operation has completed at the target window.			
21 22 23 24	MPI_RGE	result_datatype, target	dr, origin_count, origin_datatype, result_addr, result_count, t_rank, target_disp, target_count, target_datatype, op,	
25 26	IN	win, request)	initial address of huffer (choice)	
27	IN	origin_addr	initial address of buffer (choice)	
28 29	IIN	origin_count	number of entries in origin buffer (non-negative integer)	
30	IN	origin_datatype	datatype of each entry in origin buffer (handle)	
31 32	OUT	result_addr	initial address of result buffer (choice)	
33 34	IN	result_count	number of entries in result buffer (non-negative integer)	
35	IN	result_datatype	datatype of each entry in result buffer (handle)	
36 27	IN	target_rank	rank of target (non-negative integer)	
37 38 39	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	
40 41	IN	target_count	number of entries in target buffer (non-negative integer)	
42	IN	target_datatype	datatype of each entry in target buffer (handle)	
43 44	IN	ор	reduce operation (handle)	
45	IN	win	window object (handle)	
46 47	OUT	request	RMA request (handle)	
48				

```
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,
              MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
                                                                                   5
                                                                                   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
                                                                                   12
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
                                                                                   13
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                  14
    target_count
                                                                                   15
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
                                                                                   16
    result_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
                                                                                  20
    TYPE(MPI_Request), INTENT(OUT) ::
                                        request
                                                                                  21
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                        ierror
                                                                                  22
                                                                                  23
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
                                                                                  24
              RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
                                                                                  25
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
                                                                                   26
              IERROR)
                                                                                  27
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
                                                                                  28
    INTEGER(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
                                                                                  31
```

MPI\_RGET\_ACCUMULATE is similar to MPI\_GET\_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI\_RGET\_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

### 11.4 Memory Model

The memory semantics of RMA are best understood by using the concept of public and 41 private window copies. We assume that systems have a public memory region that is 42addressable 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 45process where copies of data elements from the main memory can be stored for faster access. 46Such buffers are either coherent, i.e., all updates to main memory are reflected in all private 47copies consistently, or non-coherent, i.e., conflicting accesses to main memory need to be 48

### Unofficial Draft for Comment Only

32

33

34

35

36

37 38

39

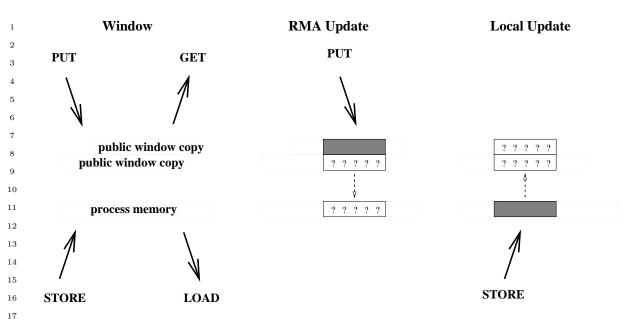


Figure 11.1: Schematic description of the public/private window operations in the MPI\_WIN\_SEPARATE memory model for two overlapping windows.

<sup>21</sup> synchronized and updated in all private copies explicitly. Coherent systems allow direct <sup>22</sup> updates to remote memory without any participation of the remote side. Non-coherent <sup>23</sup> systems, however, need to call RMA functions in order to reflect updates to the public <sup>24</sup> window in their private memory. Thus, in coherent memory, the public and the private <sup>25</sup> window are identical while they remain logically separate in the non-coherent case. MPI <sup>26</sup> thus differentiates between two memory models called *RMA unified*, if public and private <sup>27</sup> window are logically identical, and *RMA separate*, otherwise.

28In the RMA separate model, there is only one instance of each variable in process 29memory, but a distinct *public* copy of the variable for each window that contains it. A load 30 accesses the instance in process memory (this includes MPI sends). A local store accesses  $^{31}$ and updates the instance in process memory (this includes MPI receives), but the update 32 may affect other public copies of the same locations. A get on a window accesses the public 33 copy of that window. A put or accumulate on a window accesses and updates the public 34copy of that window, but the update may affect the private copy of the same locations 35 in process memory, and public copies of other overlapping windows. This is illustrated in 36 Figure **11.1**.

<sup>37</sup> In the RMA unified model, public and private copies are identical and updates via put <sup>38</sup> or accumulate calls are eventually observed by load operations without additional RMA <sup>39</sup> calls. A store access to a window is eventually visible to remote get or accumulate calls <sup>40</sup> without additional RMA calls. These stronger semantics of the RMA unified model allow <sup>41</sup> the user to omit some synchronization calls and potentially improve performance.

42

18

19 20

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.*)

#### Unofficial Draft for Comment Only

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, MPI\_GET or MPI\_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. The MPI\_WIN\_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph

#### Unofficial Draft for Comment Only

1 2

3

 $\frac{4}{5}$ 

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

 $45 \\ 46$ 

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.

33 Figure 11.2 illustrates the general synchronization pattern for active target communi-34cation. The synchronization between **post** and **start** ensures that the put call of the origin 35 process does not start until the target process exposes the window (with the **post** call); 36 the target process will expose the window only after preceding local accesses to the window 37 have completed. The synchronization between complete and wait ensures that the put call 38 of the origin process completes before the window is unexposed (with the wait call). The 39 target process will execute following local accesses to the target window only after the wait 40returned.

Figure 11.2 shows operations occurring in the natural temporal order implied by the synchronizations: the post occurs before the matching start, and complete occurs before the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as illustrated in Figure 11.3. The access to the target window is delayed until the window is exposed, after the post. However the start may complete earlier; the put and complete may also terminate earlier, if put data is buffered by the implementation. The synchronization as

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

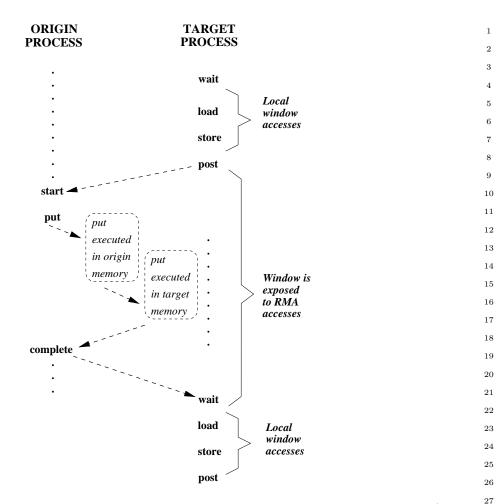


Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

*Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End of rationale.*)

 $44 \\ 45 \\ 46$ 

47 48

28

29 30

31

32

33

34

35

36

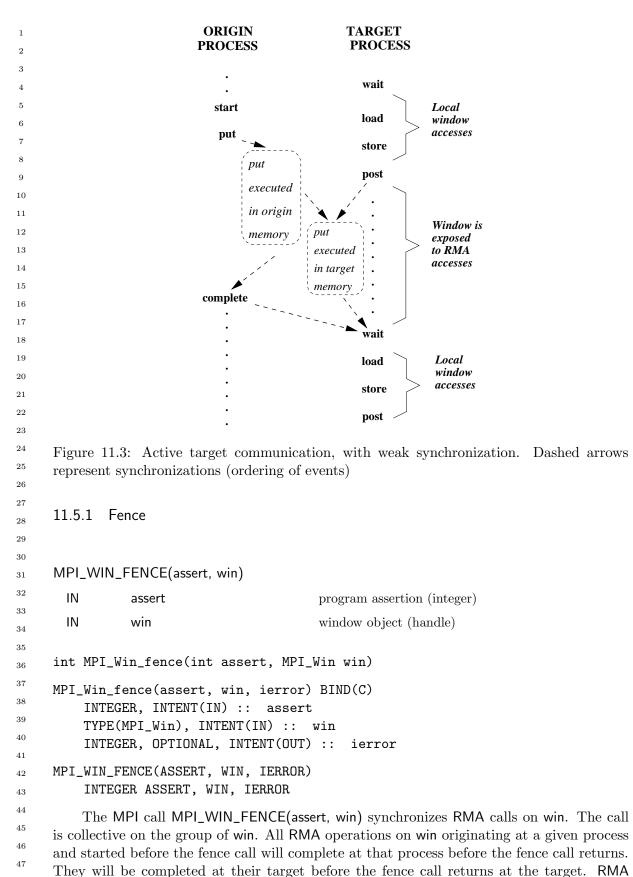
37

38 39

40

41

42



CHAPTER 11. ONE-SIDED COMMUNICATIONS

48

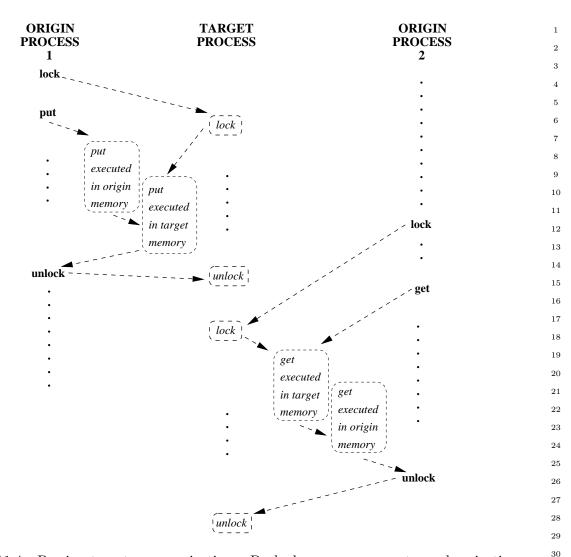


Figure 11.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

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 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 between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait.

A fence call usually entails a barrier synchronization: a process completes a call to MPI\_WIN\_FENCE only after all other processes in the group entered their matching call. However, a call to MPI\_WIN\_FENCE that is known not to end any epoch (in particular, a call with  $assert = MPI_MODE_NOPRECEDE$ ) does not necessarily act as a barrier.

 $^{31}$ 

32 33

34

35

38

40

41

42

43

44

45

1	The	e assert argument is	used to provide assertions on the context of the call that may				
2		be used for various optimizations. This is described in Section 11.5.5. A value of $assert =$					
3	0 is alwa	ays valid.					
4							
5			ls to MPI_WIN_FENCE should both precede and follow calls to				
6			functions that are synchronized with fence calls. ( $End \ of \ advice$				
7	to	users.)					
8 9	11 E O	Comoral Active To	rget Synchronization				
10	11.5.2	General Active Ta	get Synchronization				
11							
12	MPI_WI	N_START(group, as	ssert, win)				
13 14	IN	group	group of target processes (handle)				
15	IN	assert	program assertion (integer)				
16	IN	win	window object (handle)				
17			window object (number)				
18 19	int MPI	_Win_start(MPI_G	roup group, int assert, MPI_Win win)				
20	MPT Win	start(group as	sert, win, ierror) BIND(C)				
21		• •	TENT(IN) :: group				
22		EGER, INTENT(IN)					
23		E(MPI_Win), INTE					
24		EGER, OPTIONAL,					
25	MDT UTN		SERT, WIN, IERROR)				
26		EGER GROUP, ASSE					
27							
28 20			epoch for win. RMA calls issued on win during this epoch must				
29 30	access only windows at processes in group. Each process in group must issue a matching						
31			MA accesses to each target window will be delayed, if necessary, cuted the matching call to MPI_WIN_POST. MPI_WIN_START				
32		•	e corresponding MPI_WIN_POST calls are executed, but is not				
33	required						
34			used to provide assertions on the context of the call that may				
35		-	ations. This is described in Section $11.5.5$ . A value of assert =				
36		ays valid.					
37							
38 39	MPI WI	N_COMPLETE(win					
40							
41	IN	win	window object (handle)				
42	int MDT	Uin complete (MD	DT Llin				
43	int MPI	_Win_complete(MP	1_WIN WIN)				
44		_complete(win, i					
45		E(MPI_Win), INTE					
46	INT	EGER, OPTIONAL,	<pre>INTENT(OUT) :: ierror</pre>				
47 48	MPI_WIN	_COMPLETE(WIN, I	ERROR)				

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 occurred; or implementations where the first two calls are nonblocking, but the call to MPI\_WIN\_COMPLETE blocks 23until the call to MPI\_WIN\_POST occurred; or even implementations where all three calls can complete before any target process called MPI\_WIN\_POST — the data put must be 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)

			31				
IN	group	group of origin processes (handle)	32				
IN	assert	program assertion (integer)	33				
IN	win	window object (handle)	34				
			35				
int MDT 1	lin nost (MPI Crown grown	int assert, MPI_Win win)	36				
IIIC MFI_W	III_post(Mr1_droup group,	int assert, Hri_win win)	37				
MPI_Win_post(group, assert, win, ierror) BIND(C)							
TYPE(MPI_Group), INTENT(IN) :: group							
INTEGER, INTENT(IN) :: assert							
TYPE	TYPE(MPI_Win), INTENT(IN) :: win						
INTEC	INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
MDT UTN T			43				
	POST(GROUP, ASSERT, WIN,		44				
INTEC	GER GROUP, ASSERT, WIN, I	LKKUK	45				
Starts	an RMA exposure epoch for th	ne local window associated with win. Only processes	46				

Starts an RMA exposure epoch for the local window associated with win. Only processes 47in group should access the window with RMA calls on win during this epoch. Each process 48 in 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

11

12

13

1415

16

17

18

19

20

21

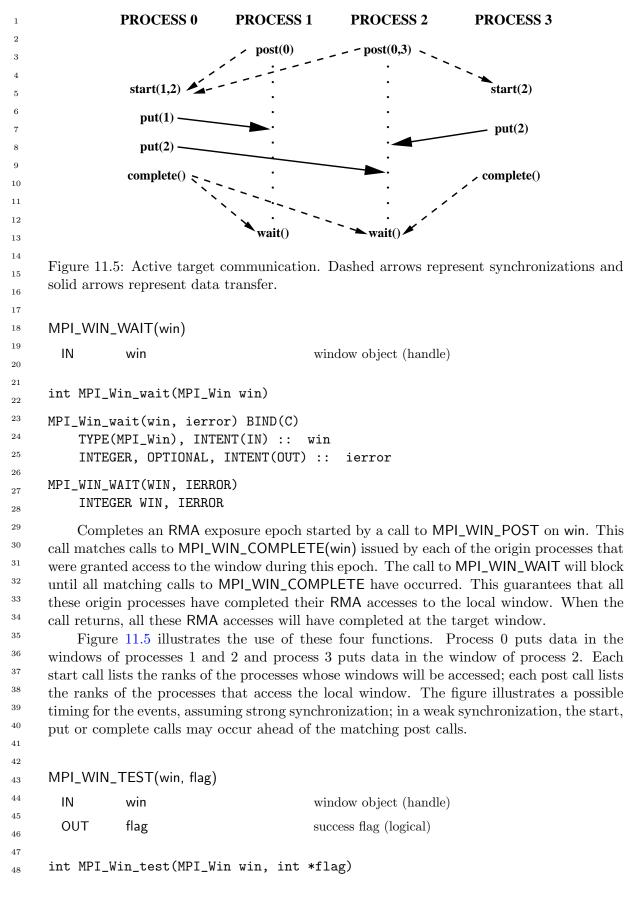
22

 $^{24}$ 

25

26

27



```
MPI_Win_test(win, flag, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
```

LOGICAL FLAG

This is the nonblocking version of MPI\_WIN\_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI\_WIN\_POST call have been completed as signalled by matching MPI\_WIN\_COMPLETE calls, and flag = false otherwise. In the former case MPI\_WIN\_WAIT would have returned immediately. The effect of return of MPI\_WIN\_TEST with flag = true is the same as the effect of a return of MPI\_WIN\_WAIT. If flag = false is returned, then the call has no visible effect.

MPI\_WIN\_TEST should be invoked only where MPI\_WIN\_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

- MPI\_WIN\_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process in group, using wincomm. No need to wait for the completion of these sends.
- MPI\_WIN\_START(group,0,win) initiate a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.
- **MPI\_WIN\_COMPLETE(win)** initiate a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.
- **MPI\_WIN\_WAIT(win)** initiate a nonblocking receive with tag tag1 from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice versa.

*Rationale.* The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a com-munication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs, in general: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (End of rationale.)

#### Unofficial Draft for Comment Only

 $^{24}$ 

1 2 3 4 5 6 7 8 9 10 11	Advice to users. Assume a communication pattern that is represented by a directed graph $G = \langle V, E \rangle$ , where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process <i>i</i> accesses the window at target process <i>j</i> . Then each process <i>i</i> issues a call to MPI_WIN_POST( <i>ingroup</i> <sub>i</sub> ,), followed by a call to MPI_WIN_START( <i>outgroup</i> <sub>i</sub> ,), where <i>outgroup</i> <sub>i</sub> = $\{j : ij \in E\}$ and <i>ingroup</i> <sub>i</sub> = $\{j : ji \in E\}$ . A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait. Note that each process may call with a group argument that has different members. ( <i>End of advice to users.</i> )						
12 13 14	11.5.3	Lock					
15 16	MPI WI	N LOCK(lock t	ype, rank, assert, win	)			
17 18	IN	lock_type	eit	, her MPI_LOCK_EXCLUSIVE or PI_LOCK_SHARED (state)			
19 20	IN	rank	rai	ık of locked window (non-negative integer)			
20	IN	assert	pro	ogram assertion (integer)			
22	IN	win	wi	ndow object (handle)			
23 24 25 26 27 28 29 30	<pre>int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win) MPI_Win_lock(lock_type, rank, assert, win, ierror) BIND(C) INTEGER, INTENT(IN) :: lock_type, rank, assert TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>						
31 32			PE, RANK, ASSERT, PE, RANK, ASSERT,				
33 34 35			ess epoch. Only the tions on win during t	window at the process with rank rank can be that epoch.			
36 37	MPI_WI	N_LOCK_ALL(a	assert, win)				
38	IN	assert	pro	ogram assertion (integer)			
39 40	IN	win	wi	ndow object (handle)			
41 42	int MPI	_Win_lock_all	(int assert, MPI_	Win win)			
43 44 45 46 47	INT TYP	EGER, INTENT( E(MPI_Win), I	sert, win, ierror) (IN) :: assert ENTENT(IN) :: win L, INTENT(OUT) ::				
48	MPI_WIN	_LOCK_ALL(ASS	ERT, WIN, IERROR)				

#### INTEGER ASSERT, WIN, IERROR

Starts an RMA access epoch to all processes in win, with a lock type of MPI\_LOCK\_SHARED. During the epoch, the calling process can access the window memory on all processes in win by using RMA operations. A window locked with MPI\_WIN\_LOCK\_ALL must be unlocked with MPI\_WIN\_UNLOCK\_ALL. This routine is not collective — the ALL refers to a lock on all members of the group of the window.

Advice to users. There may be additional overheads associated with using MPI\_WIN\_LOCK and MPI\_WIN\_LOCK\_ALL concurrently on the same window. These overheads could be avoided by specifying the assertion MPI\_MODE\_NOCHECK when possible (see Section 11.5.5). (End of advice to users.)

MPI\_WIN\_UNLOCK(rank, win)

···· ·_ ···							
IN	rank	rank of window (non-negative integer)					
IN	win	window object (handle)					
MPI_Wir INT TYF	I_Win_unlock(int rank, n_unlock(rank, win, ie FEGER, INTENT(IN) :: PE(MPI_Win), INTENT(IN FEGER, OPTIONAL, INTEN	error) BIND(C) rank 1) :: win					
	MPI_WIN_UNLOCK(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR						
Completes an RMA access epoch started by a call to MPI_WIN_LOCK(,win). RMA operations issued during this period will have completed both at the origin and at the target when the call returns.							
MPI_WI IN	IN_UNLOCK_ALL(win) win	window object (handle)					

int MPI\_Win\_unlock\_all(MPI\_Win win)
MPI\_Win\_unlock\_all(win, ierror) BIND(C)
TYPE(MPI\_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI\_WIN\_UNLOCK\_ALL(WIN, IERROR) INTEGER WIN, IERROR

Completes a shared RMA access epoch started by a call to MPI\_WIN\_LOCK\_ALL(assert, win). RMA operations issued during this epoch will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls <sup>47</sup> issued between the lock and unlock calls, and to protect load/store accesses to a locked local <sup>48</sup>

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $45 \\ 46$ 

or shared memory window executed between the lock and unlock call. Accesses that are
 protected by an exclusive lock will not be concurrent at the window site with other accesses
 to the same window that are lock protected. Accesses that are protected by a shared lock
 will not be concurrent at the window site with accesses protected by an exclusive lock to
 the same window.

<sup>6</sup> It is erroneous to have a window locked and exposed (in an exposure epoch) concur-<sup>7</sup> rently. For example, a process may not call MPI\_WIN\_LOCK to lock a target window if <sup>8</sup> the target process has called MPI\_WIN\_POST and has not yet called MPI\_WIN\_WAIT; it <sup>9</sup> 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, page 339), MPI\_WIN\_ALLOCATE (Section 11.2.2, page 407), or attached with MPI\_WIN\_ATTACH (Section 11.2.4, page 411). 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.

```
40
41 Example 11.5
```

```
<sup>42</sup> MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);
<sup>43</sup> MPI_Put(..., rank, ..., win);
<sup>44</sup> MPI_Win_unlock(rank_win);
```

44 MPI\_Win\_unlock(rank, win);
45

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

# Unofficial Draft for Comment Only

10

11

12

13

14

15

16

17 18

19

20

21 22

27

28

29

30

 $^{31}$ 

32

33

38

two calls may not block, while MPI\_WIN\_UNLOCK blocks until a lock is acquired — the update of the target window is then postponed until the call to MPI\_WIN\_UNLOCK occurs. However, if the call to MPI\_WIN\_LOCK is used to lock a local window, then the call must block until the lock is acquired, since the lock may protect local load/store accesses to the window issued after the lock call returns.

#### 11.5.4 Flush and Sync

All flush and sync functions can be called only within passive target epochs.

```
11
MPI_WIN_FLUSH(rank, win)
                                                                                          12
  IN
                                       rank of target window (non-negative integer)
           rank
                                                                                          13
                                                                                          14
  IN
           win
                                       window object (handle)
                                                                                          15
                                                                                          16
int MPI_Win_flush(int rank, MPI_Win win)
                                                                                          17
MPI_Win_flush(rank, win, ierror) BIND(C)
                                                                                          18
    INTEGER, INTENT(IN) ::
                               rank
                                                                                          19
    TYPE(MPI_Win), INTENT(IN) ::
                                      win
                                                                                          20
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                          21
                                                                                          22
MPI_WIN_FLUSH(RANK, WIN, IERROR)
                                                                                          23
    INTEGER RANK, WIN, IERROR
                                                                                          ^{24}
    MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling
                                                                                          25
process to the target rank on the specified window. The operations are completed both at
                                                                                          26
the origin and at the target.
                                                                                          27
                                                                                          28
                                                                                          29
MPI_WIN_FLUSH_ALL(win)
                                                                                          30
  IN
           win
                                       window object (handle)
                                                                                          31
                                                                                          32
                                                                                          33
int MPI_Win_flush_all(MPI_Win win)
                                                                                          34
MPI_Win_flush_all(win, ierror) BIND(C)
                                                                                          35
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                          36
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                          37
                                                                                          38
MPI_WIN_FLUSH_ALL(WIN, IERROR)
                                                                                          39
    INTEGER WIN, IERROR
```

All RMA operations issued by the calling process to any target on the specified window prior to this call and in the specified window will have completed both at the origin and at the target when this call returns.

1 2

3

4

5 6 7

8

9 10

40

41

42

```
1
     MPI_WIN_FLUSH_LOCAL(rank, win)
\mathbf{2}
       IN
                                             rank of target window (non-negative integer)
                 rank
3
       IN
                 win
                                             window object (handle)
4
5
6
     int MPI_Win_flush_local(int rank, MPI_Win win)
\overline{7}
     MPI_Win_flush_local(rank, win, ierror) BIND(C)
8
          INTEGER, INTENT(IN) :: rank
9
          TYPE(MPI_Win), INTENT(IN) ::
                                            win
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
11
12
     MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
13
          INTEGER RANK, WIN, IERROR
14
         Locally completes at the origin all outstanding RMA operations initiated by the calling
15
     process to the target process specified by rank on the specified window. For example, after
16
     this routine completes, the user may reuse any buffers provided to put, get, or accumulate
17
     operations.
18
19
20
     MPI_WIN_FLUSH_LOCAL_ALL(win)
21
       IN
                 win
                                             window object (handle)
22
23
     int MPI_Win_flush_local_all(MPI_Win win)
^{24}
25
     MPI_Win_flush_local_all(win, ierror) BIND(C)
26
          TYPE(MPI_Win), INTENT(IN) :: win
27
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
28
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
29
          INTEGER WIN, IERROR
30
^{31}
          All RMA operations issued to any target prior to this call in this window will have
32
     completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.
33
34
35
     MPI_WIN_SYNC(win)
36
       IN
                 win
                                             window object (handle)
37
38
     int MPI_Win_sync(MPI_Win win)
39
40
     MPI_Win_sync(win, ierror) BIND(C)
41
          TYPE(MPI_Win), INTENT(IN) :: win
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
43
     MPI_WIN_SYNC(WIN, IERROR)
44
          INTEGER WIN, IERROR
45
46
          The call MPI_WIN_SYNC synchronizes the private and public window copy 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 does not actually end an epoch or complete any pending MPI RMA operations).

#### 11.5.5 Assertions

The assert argument in the calls MPI\_WIN\_POST, MPI\_WIN\_START, MPI\_WIN\_FENCE, MPI\_WIN\_LOCK, and MPI\_WIN\_LOCK\_ALL is used to provide assertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program — it is erroneous to provide incorrect information. Users may always provide assert = 0 to indicate a general case where no guarantees are made.

Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent shared memory machines. Users should consult their implementation manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

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.*)

assert is the bit-vector OR of zero or more of the following integer constants: MPI\_MODE\_NOCHECK, MPI\_MODE\_NOSTORE, MPI\_MODE\_NOPUT, MPI\_MODE\_NOPRECEDE and MPI\_MODE\_NOSUCCEED. The significant options are listed below for each call.

Advice to users. C/C++ users can use bit vector or (|) to combine these constants; Fortran 90 users can use the bit-vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (End of advice to users.)

# 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.

#### Unofficial Draft for Comment Only

1

2

3 4

5

6

 $\overline{7}$ 

8

9

10

11 12

13

14

15

16

17

18 19

20

21

22 23

 $^{24}$ 

25

26

27 28

29

30

31

32

33 34

35

36

37

38

39

40

41

42 43

44

45

46 47

MPI_MODE_NOSTORE — the local window was not updated by stores (or local get or receive calls) since last synchronization. This may avoid the need for cache 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 con- flicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.
Advice to users. Note that the nostore and noprecede flags provide information on what happened <i>before</i> the call; the noput and nosucceed flags provide information on what will happen <i>after</i> the call. ( <i>End of advice to users.</i> )
11.5.6 Miscellaneous Clarifications
Once an RMA routine completes, it is safe to free any opaque objects passed as argument to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete. As in message-passing, datatypes must be committed before they can be used in RMA 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, page 342).

CHAPTER 11. ONE-SIDED COMMUNICATIONS

# 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.

MPI_ERR_WIN	invalid win argument	6
MPI_ERR_BASE	invalid base argument	7
MPI_ERR_SIZE	invalid size argument	8
MPI_ERR_DISP	invalid disp argument	9
MPI_ERR_LOCKTYPE	invalid locktype argument	10
MPI_ERR_ASSERT	invalid assert argument	11
MPI_ERR_RMA_CONFLICT	conflicting accesses to window	12
MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls	13
MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case	14
	of a window created with	15
	MPI_WIN_CREATE_DYNAMIC, target memory is not	16
	attached)	17
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource	18
	exhaustion)	19
MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the	20
	group of the specified communicator cannot expose	21
	shared memory)	22
MPI_ERR_RMA_WRONG_FLAVOR	passed window has the wrong flavor for the called	23
	function	24
		25

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.

Unofficial Draft for Comment Only

	454	CHAPTER 11. ONE-SIDED COMMUNICATIONS
1 2 3	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 5 6 7 8	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.
9 10 11 12 13 14 15	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.
16 17 18 19 20 21 22 23	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.
23 24		The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public
25 26		to private copy $(6)$ is the same call that completes the put or accumulate operation in vindow copy $(2, 3)$ . If a put or accumulate access was synchronized with a lock, then
27		update of the public window copy is complete as soon as the updating process executed
28		_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the
29		ate of private copy in the process memory may be delayed until the target process
30		utes a synchronization call on that window (6). Thus, updates to process memory can
31	-	ys be delayed in the RMA separate memory model until the process executes a suitable
32		hronization call, while they must complete in the RMA unified model without additional
33	•	hronization calls. If fence or post-start-complete-wait synchronization is used, updates
34		public window copy can be delayed in both memory models until the window owner
35		utes a synchronization call. When passive-target synchronization (lock/unlock or even
36	пush	) is used, it is necessary to update the public window copy in the RMA separate model,

or the private window copy in the RMA unified model, even if the window owner does not execute any related synchronization call. The rules above also define, by implication, when an update to a public window copy

becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI\_WIN\_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI\_WIN\_FENCE(0, win2) makes these updates visible in the public copy of win2. 

The behavior of some MPI RMA operations may be *undefined* in 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 

#### **Unofficial Draft for Comment Only**

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.

As discussed in [6], requiring operations such as overlapping puts to be Rationale. erroneous makes it difficult to use MPI RMA to implement programming models such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any implementation that enforces this rule, as it would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (End of rationale.)

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not generate an MPI exception. (End of advice to implementors.)

A program with a well-defined outcome in the MPI\_WIN\_SEPARATE memory model must obey the following rules.

- 1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- 2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public 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.
- 3. A put or accumulate must not access a target window once a load/store update or a 35 put or accumulate update to another (overlapping) target window has started on a 36 location in the target window, until the update becomes visible in the public copy of 37 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. 41

The last constraint on correct RMA accesses may seem unduly restric-Rationale. 43 tive, as it forbids concurrent accesses to nonoverlapping locations in a window. The 44reason for this constraint is that, on some architectures, explicit coherence restoring 45operations may be needed at synchronization points. A different operation may be 46needed for locations that were updated by stores and for locations that were remotely 47updated by put or accumulate operations. Without this constraint, the MPI library 48

#### **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

9

10

11

12

13

14

1516

17

18

19

2021

22

23 $^{24}$ 

25

26

2728

2930

 $^{31}$ 

32 33

34

38

39

40

will have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*End of rationale.*)

Note that MPI\_WIN\_SYNC may be used within a passive target epoch to synchronize the 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 and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with a well-defined outcome must obey in this case are:

10 11

1

2

3

4

5

6

7

8 9

12 13 14

26

27

28

29

30

31

32 33

34

35

36

37

38

39

40

41

42

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.

152. Accessing a location in the window that is also the target of a remote update is valid 16(not erroneous) but the precise result will depend on the behavior of the implemen-17 tation. Updates from a remote process will appear in the memory of the target, but 18 there are no atomicity or ordering guarantees if more than one byte is updated. Up-19dates are stable in the sense that once data appears in memory of the target, the data 20remains until replaced by another update. This permits polling on a location for a 21change from zero to non-zero or for a particular value, but not polling and comparing 22 the relative magnitude of values. Users are cautioned that polling on one memory 23location and then accessing a different memory location has defined behavior only if 24the other rules given here and in this chapter are followed. 25

> 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 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.

Advice to users. A user can write correct programs by following the following rules:

- **fence:** During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- **post-start-complete-wait:** A window should not be updated with store operations while being posted, if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

- **lock:** Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for load/store accesses and for RMA accesses.
- changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two overlapping 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 RMA accesses to the window are synchronized with fences; after a local call to MPI\_WIN\_WAIT, if the accesses are synchronized with post-start-completewait; after the call at the origin (local or remote) to MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10 11

12 13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28 29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40 41

42

43

44

45

46

```
1
         The semantics are illustrated by the following examples:
\mathbf{2}
     Example 11.6 The following example demonstrates updating a memory location inside
3
     a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK
4
     and MPI_WIN_UNLOCK calls around the store to X in process B are necessary to ensure
5
     consistency between the public and private copies of the window.
6
7
     Process A:
                                    Process B:
8
                                    window location X
9
10
                                    MPI_Win_lock(EXCLUSIVE,B)
11
                                    store X /* local update to private copy of B */
12
                                    MPI_Win_unlock(B)
13
                                    /* now visible in public window copy */
14
15
     MPI_Barrier
                                    MPI_Barrier
16
17
     MPI_Win_lock(EXCLUSIVE,B)
18
     MPI_Get(X) /* ok, read from public window */
19
     MPI_Win_unlock(B)
20
21
     Example 11.7 In the RMA unified model, although the public and private copies of the
22
     windows are synchronized, caution must be used when combining load/stores and multi-
23
     process synchronization. Although the following example appears correct, the compiler or
24
     hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET
25
     returning the incorrect value of X.
26
27
     Process A:
                                    Process B:
28
                                    window location X
29
30
                                    store X /* update to private&public copy of B */
^{31}
     MPI_Barrier
                                    MPI_Barrier
32
     MPI_Win_lock_all
33
     MPI_Get(X) /* ok, read from window */
34
     MPI_Win_flush_local(B)
35
     /* read value in X */
36
     MPI_Win_unlock_all
37
38
     MPI_BARRIER provides process synchronization, but not memory synchronization. The
39
     example could potentially be made safe through the use of compiler and hardware specific
40
     notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The
41
     use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct
42
     result.
43
44
     Example 11.8 The following example demonstrates the reading of a memory location
45
     updated by a remote process (Rule 6) in the RMA separate memory model. Although the
```

<sup>46</sup> MPI\_WIN\_UNLOCK on process A and the MPI\_BARRIER ensure that the public copy on
 <sup>47</sup> process B reflects the updated value of X, the call to MPI\_WIN\_LOCK by process B is
 <sup>48</sup> necessary to synchronize the private copy with the public copy.

Process A:	Process B:	1
TIOCESS A.	window location X	2
		3
MPI_Win_lock(EXCLUSIVE,B)		4
MPI_Put(X) /* update to pu	ublic window */	5
MPI_Win_unlock(B)		6
		7
MPI_Barrier	MPI_Barrier	8
		9
	MPI_Win_lock(EXCLUSIVE,B)	10
	<pre>/* now visible in private copy of B */</pre>	11
	load X	12
	MPI_Win_unlock(B)	13
Note that in this example th	e barrier is not critical to the semantic correctness. The	14 15
	ees a remote process will not modify the public copy after	15
_	the private and public copies. A polling implementation	10
-	cess B would be semantically correct. The barrier is required	18
0 0 1	rms the put operation before process B performs the load of	19
Х.		20
		21
-	mple 11.7, the following example is unsafe even in the unified	22
	n not be guaranteed to occur after the MPI_BARRIER. While	23
	xplicitly synchronize the public and private copies through	24
	PUT will update both the public and private copies of the	25
	oad could result in old values of X being returned. Compiler	26
and hardware specific notations could ensure the load occurs after the data is updated, explicit one-sided synchronization calls can be used to ensure the proper result.		
explicit one-sided synchronizat	ion cans can be used to ensure the proper result.	28
Process A:	Process B:	29
	window location X	30
MPI_Win_lock_all		31
MPI_Put(X) /* update to wi	indow */	32
MPI_Win_flush(B)		33
		34 35
MPI_Barrier	MPI_Barrier	36
	load X	37
MPI_Win_unlock_all		38
		39
Example 11.10 The followi	ng example further clarifies Rule 5. MPI_WIN_LOCK and	40
-	update the public copy of a window with changes to the	41
	is no guarantee that process A in the following sequence will	42
	by the local store by process B before the lock.	43
*		44
Process A:	Process B:	45

window	loca	ation X							
store X	: /*	update	to	private	сору	of	В	*/	

**Unofficial Draft for Comment Only** 

```
1
                                    MPI_Win_lock(SHARED,B)
\mathbf{2}
                                    MPI_Barrier
     MPI_Barrier
3
4
     MPI_Win_lock(SHARED,B)
\mathbf{5}
     MPI_Get(X) /* X may be the X before the store */
6
     MPI_Win_unlock(B)
7
                                    MPI_Win_unlock(B)
8
                                    /* update on X now visible in public window */
9
     The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would
10
     guarantee process A would see the updated value of X, as the public copy of the window
11
     would be explicitly synchronized with the private copy.
12
13
     Example 11.11 Similar to the previous example, Rule 5 can have unexpected implications
14
     for general active target synchronization with the RMA separate memory model. It is not
15
     guaranteed that process B reads the value of X as per the local update by process A, because
16
     neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in
17
     the public window copy.
18
19
     Process A:
                                    Process B:
20
     window location X
21
     window location Y
22
23
     store Y
^{24}
     MPI_Win_post(A,B) /* Y visible in public window */
25
     MPI_Win_start(A)
                                    MPI_Win_start(A)
26
27
     store X /* update to private window */
28
29
     MPI_Win_complete
                                    MPI_Win_complete
30
     MPI_Win_wait
^{31}
     /* update on X may not yet visible in public window */
32
33
     MPI_Barrier
                                    MPI_Barrier
34
35
                                    MPI_Win_lock(EXCLUSIVE,A)
36
                                    MPI_Get(X) /* may return an obsolete value */
37
                                    MPI_Get(Y)
38
                                    MPI_Win_unlock(A)
39
40
     To allow process B to read the value of X stored by A the local store must be replaced by
41
     a local MPI_PUT that updates the public window copy. Note that by this replacement X
42
     may become visible in the private copy of process A only after the MPI_WIN_WAIT call in
43
     process A. The update to Y made before the MPI_WIN_POST call is visible in the public
44
     window after the MPI_WIN_POST call and therefore process B will read the proper value
45
     of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START
46
     operation, and process B would still get the value stored by process A.
47
```

460

**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.

Process A:	Process B: window location X	
<pre>MPI_Win_lock(EXCLUSIVE,B) MPI_Put(X) /* update to pu MPI_Win_unlock(B)</pre>	blic window */	
MPI_Barrier	MPI_Barrier	
	MPI_Win_post(B) MPI_Win_start(B)	
	<pre>load X /* access to private window */     /* may return an obsolete value */</pre>	
	MPI_Win_complete MPI_Win_wait	

To ensure that the value put by process A is read, the local load must be replaced with a local MPI\_GET operation, or must be placed after the call to MPI\_WIN\_WAIT.

# 11.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply, see the info key accumulate\_ops in Section 11.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation cannot be accessed by a load or an RMA call other than accumulate until the accumulate operation has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

# 11.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. <sup>43</sup> MPI specifies ordering between accumulate operations from one process to the same (or overlapping) memory locations at another process on a per-datatype granularity. The default ordering is strict ordering, which guarantees that overlapping updates from the same source to a remote location are committed in program order and that reads (e.g., with MPI\_GET\_ACCUMULATE) and writes (e.g., with MPI\_ACCUMULATE) are executed and <sup>43</sup>

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

committed in program order. Ordering only applies to operations originating at the same
 origin that access overlapping target memory regions. MPI does not provide any guarantees
 for accesses or updates from different origins to overlapping target memory regions.

4 The default strict ordering may incur a significant performance penalty. MPI specifies  $\mathbf{5}$ the info key accumulate\_ordering to allow relaxation of the ordering semantics when specified 6 to any window creation function. The values for this key are as follows. If set to none,  $\overline{7}$ then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 8 in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list of 9 required access orderings at the target. Allowed values in the comma-separated list are rar, 10 war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-after-write 11ordering, respectively. These indicate whether operations of the specified type complete in 12the order they were issued. For example, raw means that any writes must complete at the 13target before any reads. These ordering requirements apply only to operations issued by 14the same origin process and targeting the same target process. The default value for 15accumulate\_ordering is rar, raw, war, waw, which implies that writes complete at the target in the 16order in which they were issued, reads complete at the target before any writes that are 17issued after the reads, and writes complete at the target before any reads that are issued after 18 the writes. Any subset of these four orderings can be specified. For example, if only read-19after-read and write-after-write ordering is required, then the value of the accumulate\_ordering 20key 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

23 24

25

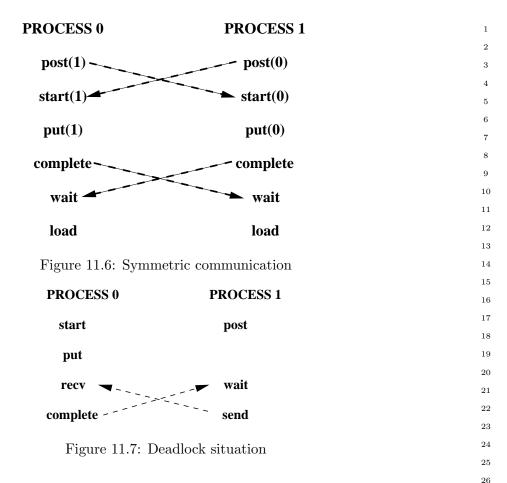
One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication 29becomes enabled. This fuzziness provides to the implementor more flexibility than with 30 point-to-point communication. Access to a target window becomes enabled once the corre- $^{31}$ sponding synchronization (such as MPI\_WIN\_FENCE or MPI\_WIN\_POST) has executed. On 32 the origin process, an RMA communication may become enabled as soon as the correspond-33 ing put, get or accumulate call has executed, or as late as when the ensuing synchronization 34call is issued. Once the communication is enabled both at the origin and at the target, the 35 communication must complete. 36

Consider the code fragment in Example 11.4, on page 443. 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.

41 Consider the code fragment in Example 11.5, on page 448. Some of the calls may block
 42 if another process holds a conflicting lock. However, if no conflicting lock is held, then the
 43 code fragment must complete.

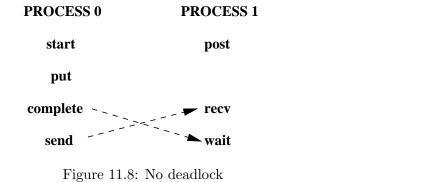
Consider the code illustrated in Figure 11.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication



should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock, if the order of the complete and wait calls is reversed, at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete



Unofficial Draft for Comment Only

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41 42

43 44

 $45 \\ 46$ 

47

on process 0 can proceed to completion. Process 0 will reach the send call, allowing the  $\mathbf{2}$ receive call of process 1 to complete.

3 4

56

7

8 9

27

28

2930

31

32

33

34

35

36

37

38 39

1

Rationale. MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true 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.

10 A similar issue is whether such progress must occur while a process is busy comput-11 ing, or blocked in a non-MPI call. Suppose that in the last example the send-receive 12pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not spec-13 ify whether deadlock is avoided. Suppose that the blocking receive of process 1 is 14replaced by a very long compute loop. Then, according to one interpretation of the 15MPI standard, process 0 must return from the complete call after a bounded delay, 16even if process 1 does not reach any MPI call in this period of time. According to 17 another interpretation, the complete call may block until process 1 reaches the wait 18 call, or reaches another MPI call. The qualitative behavior is the same, under both 19 interpretations, unless a process is caught in an infinite compute loop, in which case 20the difference may not matter. However, the quantitative expectations are different. 21Different MPI implementations reflect these different interpretations. While this am-22biguity is unfortunate, it does not seem to affect many real codes. The MPI Forum 23decided not to decide which interpretation of the standard is the correct one, since the  $^{24}$ issue is very contentious, and a decision would have much impact on implementors 25but less impact on users. (*End of rationale.*) 26

#### **Registers and Compiler Optimizations** 11.7.4

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI\_WIN\_UNIFIED.

The problem is illustrated by the following code:

40	Source of Process 1	Source of Process 2	Executed in Process 2
41	bbbb = 777	buff = 999	reg_A:=999
42	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
43	call MPI_PUT(bbbb		stop appl.thread
44	into buff of process 2)		buff:=777 in PUT handler
45			continue appl.thread
46	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
47		ccc = buff	ccc:=reg_A
48			

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 compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in 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, especially in Sections 17.1.12 and 17.1.13 on pages 628-630 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 17.1.16 12to 17.1.19 on pages 633 to 643 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". Sections "Solutions" to "VOLATILE" on pages 636-641 discuss several solutions for the problem in this example.

#### Examples 11.8

**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.

```
. . .
while(!converged(A)){
 update(A);
 MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
 for(i=0; i < toneighbors; i++)</pre>
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                          todisp[i], 1, totype[i], win);
 MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
 }
```

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 in two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither use nor provide communicated data, is updated.

# **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10 11

13

1415

16

1718

192021

22

23

 $^{24}$ 25

26

27

28 29

30

31

32

33

34

35

36

37

38 39

40

41

42

```
1
     . . .
\mathbf{2}
     while(!converged(A)){
3
       update_boundary(A);
4
       MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
5
       for(i=0; i < fromneighbors; i++)</pre>
6
          MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
7
                            fromdisp[i], 1, fromtype[i], win);
8
       update_core(A);
9
       MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
10
       }
11
     The get communication can be concurrent with the core update, since they do not access the
12
     same locations, and the local update of the origin buffer by the get call can be concurrent
13
     with the local update of the core by the update_core call. In order to get similar overlap
14
     with put communication we would need to use separate windows for the core and for the
15
     boundary. This is required because we do not allow local stores to be concurrent with puts
16
     on the same, or on overlapping, windows.
17
18
     Example 11.15 Same code as in Example 11.13, rewritten using post-start-complete-wait.
19
20
21
     while(!converged(A)){
22
       update(A);
23
       MPI_Win_post(fromgroup, 0, win);
24
       MPI_Win_start(togroup, 0, win);
25
       for(i=0; i < toneighbors; i++)</pre>
26
          MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
27
                                  todisp[i], 1, totype[i], win);
28
       MPI_Win_complete(win);
29
       MPI_Win_wait(win);
30
       }
^{31}
32
     Example 11.16 Same example, with split phases, as in Example 11.14.
33
34
35
     while(!converged(A)){
36
       update_boundary(A);
37
       MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
38
       MPI_Win_start(fromgroup, 0, win);
39
       for(i=0; i < fromneighbors; i++)</pre>
40
          MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
41
                           fromdisp[i], 1, fromtype[i], win);
42
       update_core(A);
43
       MPI_Win_complete(win);
44
       MPI_Win_wait(win);
45
       }
46
47
48
```

**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.

```
. . .
if (!converged(A0,A1))
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
/* the barrier is needed because the start call inside the
                                                                                    10
loop uses the nocheck option */
                                                                                    11
while(!converged(A0, A1)){
                                                                                    12
  /* communication on AO and computation on A1 */
                                                                                    13
  update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
                                                                                    14
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
                                                                                    15
  for(i=0; i < fromneighbors; i++)</pre>
                                                                                    16
    MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
                                                                                    17
                fromdisp0[i], 1, fromtype0[i], win0);
                                                                                    18
  update1(A1); /* local update of A1 that is
                                                                                    19
                   concurrent with communication that updates A0 */
                                                                                    20
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
                                                                                    21
  MPI_Win_complete(win0);
                                                                                    22
  MPI_Win_wait(win0);
                                                                                    23
                                                                                    ^{24}
  /* communication on A1 and computation on A0 */
                                                                                    25
  update2(A0, A1); /* local update of A0 that depends on A1 (and A0)*/
                                                                                    26
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
                                                                                    27
  for(i=0; i < fromneighbors; i++)</pre>
                                                                                    28
    MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                                                                                    29
                 fromdisp1[i], 1, fromtype1[i], win1);
                                                                                    30
  update1(A0); /* local update of A0 that depends on A0 only,
                                                                                    31
                  concurrent with communication that updates A1 */
                                                                                    32
  if (!converged(A0,A1))
                                                                                    33
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
                                                                                    34
  MPI_Win_complete(win1);
                                                                                    35
  MPI_Win_wait(win1);
                                                                                    36
  }
                                                                                    37
```

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.

44Put 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

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4 5

6

7

8

9

38 39

40 41

42

43

4546

47

```
1
     z = MPI_Get_accumulate(...)
\mathbf{2}
     means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr
3
     in the description of MPI_GET_ACCUMULATE) on the left side of the assignment; in this
4
     case, z. This format is also used with MPI_COMPARE_AND_SWAP.
5
6
     Example 11.18 The following example implements a naive, non-scalable counting sema-
7
     phore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy
8
     of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch
9
     opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the
10
     public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE
11
     are used to write to or read from the local public copy.
12
13
     Process A:
                                                    Process B:
14
                                                    MPI_Win_lock_all
     MPI_Win_lock_all
15
     window location X
16
     X=2
17
     MPI_Win_sync
18
     MPI_Barrier
                                                    MPI_Barrier
19
20
     MPI_Accumulate(X, MPI_SUM, -1)
                                                    MPI_Accumulate(X, MPI_SUM, -1)
21
22
     stack variable z
                                                     stack variable z
23
     do
                                                     do
^{24}
       z = MPI_Get_accumulate(X,
                                                       z = MPI_Get_accumulate(X,
25
             MPI_NO_OP, 0)
                                                            MPI_NO_OP, 0)
26
       MPI_Win_flush(A)
                                                       MPI_Win_flush(A)
27
     while(z!=0)
                                                     while(z!=0)
28
29
     MPI_Win_unlock_all
                                                    MPI_Win_unlock_all
30
^{31}
     Example 11.19 Implementing a critical region between two processes (Peterson's al-
32
     gorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and
33
     MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared
34
     access epochs to all processes from all other processes in a window. Once the access epochs
35
     are established, accumulate communication operations and flush and sync synchronization
36
     operations can be used to read from or write to the public copy of the window.
37
38
                                                Process B:
     Process A:
39
     window location X
                                                window location Y
40
     window location T
41
42
     MPI_Win_lock_all
                                                MPI_Win_lock_all
43
     X=1
                                                Y=1
44
     MPI_Win_sync
                                                MPI_Win_sync
45
     MPI_Barrier
                                                MPI_Barrier
46
     MPI_Accumulate(T, MPI_REPLACE, 1)
                                                MPI_Accumulate(T, MPI_REPLACE, 0)
47
     stack variables t,y
                                                stack variable t,x
48
     t=1
                                                t=0
```

468

y=MPI_Get_accumulate(Y,	x=MPI_Get_accumulate(X,	1
MPI_NO_OP, 0)	MPI_NO_OP, 0)	2
while(y==1 && t==1) do	while(x==1 && t==0) do	3
<pre>y=MPI_Get_accumulate(Y,</pre>	x=MPI_Get_accumulate(X,	4
MPI_NO_OP, 0)	MPI_NO_OP, 0)	5
t=MPI_Get_accumulate(T,	t=MPI_Get_accumulate(T,	6
MPI_NO_OP, 0)	MPI_NO_OP, 0)	7
MPI_Win_flush_all	MPI_Win_flush(A)	8
done	done	9
<pre>// critical region</pre>	// critical region	10
<pre>MPI_Accumulate(X, MPI_REPLACE, 0)</pre>	<pre>MPI_Accumulate(Y, MPI_REPLACE, 0)</pre>	11
MPI_Win_unlock_all	MPI_Win_unlock_all	12

**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.

Process A:	Process B:	21
MPI_Win_lock_all	MPI_Win_lock_all	22
atomic location A		23
A=0		24
MPI_Win_sync		25
MPI_Barrier	MPI_Barrier	26
stack variable r=1	<pre>stack variable r=1</pre>	27
while(r != 0) do	while(r != 0) do	28
<pre>r = MPI_Compare_and_swap(A, 0, 1)</pre>	<pre>r = MPI_Compare_and_swap(A, 0, 1)</pre>	29
MPI_Win_flush(A)	MPI_Win_flush(A)	30
done	done	31
// critical region	// critical region	32
r = MPI_Compare_and_swap(A, 1, 0)	r = MPI_Compare_and_swap(A, 1, 0)	33
MPI_Win_unlock_all	MPI_Win_unlock_all	34

**Example 11.21** The following example shows how request-based operations can be used to overlap communication with computation. Each process fetches, processes, and writes the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to allow up to M communication operations to overlap with computation.

```
41
int
             i, j;
                                                                                         42
MPI_Win
             win;
MPI_Request put_req[M] = { MPI_REQUEST_NULL };
                                                                                         43
                                                                                         44
MPI_Request get_req;
double
                                                                                         45
             **baseptr;
                                                                                         46
double
             data[M][N];
                                                                                         47
```

MPI\_Win\_allocate(NSTEPS\*N\*sizeof(double), sizeof(double), MPI\_INFO\_NULL,

```
1
       MPI_COMM_WORLD, baseptr, &win);
\mathbf{2}
3
     MPI_Win_lock_all(0, win);
4
\mathbf{5}
     for (i = 0; i < NSTEPS; i++) {</pre>
6
       if (i<M)
7
         j=i;
8
      else
9
        MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
10
11
      MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
12
                 &get_req);
13
      MPI_Wait(&get_req,MPI_STATUS_IGNORE);
14
      compute(i, data[j], ...);
15
      MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
16
                 &put_req[j]);
17
     }
18
19
     MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
20
     MPI_Win_unlock_all(win);
21
22
     Example 11.22 The following example constructs a distributed shared linked list using
23
     dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
24
     and broadcasts the pointer to all processes. All processes then concurrently append N new
25
     elements to the list. When a process attempts to attach its element to the tail of the
26
     list it may discover that its tail pointer is stale and it must chase ahead to the new tail
27
     before the element can be attached. This example requires some modification to work in
28
     an environment where the length of a pointer is different on different processes.
29
30
     . . .
^{31}
     #define NUM_ELEMS 10
32
33
     /* Linked list pointer */
34
     typedef struct {
35
       MPI_Aint disp;
36
       int
                  rank;
37
     } llist_ptr_t;
38
39
     /* Linked list element */
40
     typedef struct {
41
       llist_ptr_t next;
42
       int value;
43
     } llist_elem_t;
44
45
     const llist_ptr_t nil = { -1, (MPI_Aint) MPI_BOTTOM };
46
47
     /* List of locally allocated list elements. */
48
```

```
1
static llist_elem_t **my_elems = NULL;
                                                                                    \mathbf{2}
static int my_elems_size = 0;
                                                                                    3
static int my_elems_count = 0;
                                                                                    4
/* Allocate a new shared linked list element */
                                                                                    5
MPI_Aint alloc_elem(int value, MPI_Win win) {
                                                                                    6
                                                                                    7
  MPI_Aint disp;
                                                                                     8
  llist_elem_t *elem_ptr;
                                                                                    9
                                                                                    10
  /* Allocate the new element and register it with the window */
                                                                                    11
  MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
  elem_ptr->value = value;
                                                                                    12
                                                                                    13
  elem_ptr->next = nil;
                                                                                    14
  MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
                                                                                    15
                                                                                    16
  /* Add the element to the list of local elements so we can free
                                                                                    17
     it later. */
                                                                                    18
  if (my_elems_size == my_elems_count) {
                                                                                    19
    my_elems_size += 100;
                                                                                    20
    my_elems = realloc(my_elems, my_elems_size);
                                                                                    21
  }
  my_elems[my_elems_count] = elem_ptr;
                                                                                    22
                                                                                    23
  my_elems_count++;
                                                                                    24
                                                                                    25
  MPI_Get_address(elem_ptr, &disp);
                                                                                    26
  return disp;
}
                                                                                    27
                                                                                    28
                                                                                    29
int main(int argc, char *argv[]) {
                                                                                    30
  int
                procid, nproc, i;
                                                                                    31
  MPI_Win
                llist_win;
                                                                                    32
                head_ptr, tail_ptr;
  llist_ptr_t
                                                                                    33
                                                                                    34
  MPI_Init(&argc, &argv);
                                                                                    35
  MPI_Comm_rank(MPI_COMM_WORLD, &procid);
                                                                                    36
                                                                                    37
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
                                                                                    38
                                                                                    39
  MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
                                                                                    40
                                                                                    41
  /* Process 0 creates the head node */
                                                                                    42
  if (procid == 0)
    head_ptr.disp = alloc_elem(-1, llist_win);
                                                                                    43
                                                                                    44
  /* Broadcast the head pointer to everyone */
                                                                                    45
                                                                                    46
  head_ptr.rank = 0;
                                                                                    47
  MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
                                                                                    48
  tail_ptr = head_ptr;
```

```
1
2
       /* Lock the window for shared access to all targets */
3
       MPI_Win_lock_all(0, llist_win);
4
5
       /* All processes concurrently append NUM_ELEMS elements to the list */
6
       for (i = 0; i < NUM_ELEMS; i++) {</pre>
7
         llist_ptr_t new_elem_ptr;
8
         int success;
9
10
         /* Create a new list element and attach it to the window */
11
         new_elem_ptr.rank = procid;
12
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
13
14
         /* Append the new node to the list. This might take multiple
15
            attempts if others have already appended and our tail pointer
            is stale. */
16
17
         do {
18
           llist_ptr_t next_tail_ptr = nil;
19
20
           MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
21
                (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
22
                (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.rank),
23
               llist_win);
24
25
           MPI_Win_flush(tail_ptr.rank, llist_win);
26
           success = (next_tail_ptr.rank == nil.rank);
27
28
           if (success) {
29
             MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
30
                  (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.disp), 1,
31
                 MPI_AINT, MPI_REPLACE, llist_win);
32
33
             MPI_Win_flush(tail_ptr.rank, llist_win);
34
             tail_ptr = new_elem_ptr;
35
36
           } else {
37
             /* Tail pointer is stale, fetch the displacement.
                                                                   May take
38
                multiple tries if it is being updated. */
39
             do {
40
               MPI_Get_accumulate( NULL, 0, MPI_AINT, &next_tail_ptr.disp,
41
                    1, MPI_AINT, tail_ptr.rank,
42
                    (MPI_Aint) &(((llist_elem_t*)tail_ptr.disp)->next.disp),
43
                    1, MPI_AINT, MPI_NO_OP, llist_win);
44
45
               MPI_Win_flush(tail_ptr.rank, llist_win);
46
             } while (next_tail_ptr.disp == nil.disp);
47
             tail_ptr = next_tail_ptr;
48
           }
```

```
} while (!success);
                                                                                       1
 }
                                                                                       \mathbf{2}
                                                                                       3
                                                                                       4
 MPI_Win_unlock_all(llist_win);
 MPI_Barrier( MPI_COMM_WORLD );
                                                                                       5
                                                                                       6
 /* Free all the elements in the list */
                                                                                       7
 for ( ; my_elems_count > 0; my_elems_count--) {
                                                                                       8
    MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
                                                                                       9
                                                                                       10
   MPI_Free_mem(my_elems[my_elems_count-1]);
                                                                                       11
 }
                                                                                       12
 MPI_Win_free(&llist_win);
                                                                                       13
. . .
                                                                                       14
                                                                                       15
                                                                                       16
                                                                                       17
                                                                                       18
                                                                                       19
                                                                                       20
                                                                                       21
```

```
473
```

## 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.*)

1 2 3 4 5 6 7	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- plication. For a generalized request, the operation associated with the request is per- formed by the application; therefore, the application must notify MPI through a call to MPI_GREQUEST_COMPLETE when the operation completes. MPI maintains the "comple- tion" status of generalized requests. Any other request state has to be maintained by the					
7 8 9 10	user. A new generalized request is started with					
11	MPI_GRE	QUEST_START(query_fr	, free_fn, cancel_fn, extra_state, request)			
12 13	IN	query_fn	callback function invoked when request status is queried (function)			
14 15 16	IN	free_fn	callback function invoked when request is freed (function)			
17 18	IN	cancel_fn	callback function invoked when request is cancelled (function)			
19	IN	extra_state	extra state			
20 21	OUT	request	generalized request (handle)			
24 25 26 27	<pre>MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state, MPI_Request *request)</pre>					
28 29 30 31 32	<pre>MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,</pre>					
33		GER(KIND=MP1_ADDRESS_ (MPI_Request), INTENT	KIND), INTENT(IN) :: extra_state (OUT) :: request			
34		GER, OPTIONAL, INTENT	-			
35 36 37	MPI_GREQU	JEST_START(QUERY_FN, IERROR)	FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,			
38	INTEC	GER REQUEST, IERROR				
39	EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN					
40	INTEC	GER (KIND=MPI_ADDRESS	S_KIND) EXTRA_STATE			
41 42 43 44		<i>ice to users.</i> Note that ests, in C and Fortran. (	a generalized request is of the same type as regular End of advice to users.)			
45 46 47 48	The s	yntax and meaning of th	equest and returns a handle to it in <b>request</b> . ne callback functions are listed below. All callback func- argument that was associated with the request by the			

```
1
starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined
                                                                                           \mathbf{2}
state for the request.
                                                                                           3
    In C, the query function is
                                                                                           4
typedef int MPI_Grequest_query_function(void *extra_state,
                                                                                           5
               MPI_Status *status);
                                                                                           6
in Fortran with the mpi_f08 module
                                                                                           7
ABSTRACT INTERFACE
                                                                                           9
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
                                                                                          10
  BIND(C)
                                                                                          11
       TYPE(MPI_Status) :: status
                                                                                          12
       INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                          13
       INTEGER :: ierror
                                                                                          14
in Fortran with the mpi module and mpif.h
                                                                                          15
                                                                                          16
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
                                                                                          17
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
                                                                                          18
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          19
                                                                                          20
    The query_fn function computes the status that should be returned for the generalized
                                                                                          21
request. The status also includes information about successful/unsuccessful cancellation of
                                                                                          22
the request (result to be returned by MPI_TEST_CANCELLED).
                                                                                          23
    The query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that
                                                                                          ^{24}
completed the generalized request associated with this callback. The callback function is
                                                                                          25
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when
                                                                                          26
the call occurs. In both cases, the callback is passed a reference to the corresponding
                                                                                          27
status variable passed by the user to the MPI call; the status set by the callback function
                                                                                          28
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or
                                                                                          29
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI
                                                                                          30
will pass a valid status object to query_fn, and this status will be ignored upon return of the
                                                                                          31
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE
                                                                                          32
is called on the request; it may be invoked several times for the same generalized request,
                                                                                          33
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also
                                                                                          34
that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn
                                                                                          35
callback functions, one for each generalized request that is completed by the MPI call. The
                                                                                          36
order of these invocations is not specified by MPI.
                                                                                          37
    In C, the free function is
                                                                                          38
                                                                                          39
typedef int MPI_Grequest_free_function(void *extra_state);
                                                                                          40
in Fortran with the mpi_f08 module
                                                                                          41
                                                                                          42
ABSTRACT INTERFACE
                                                                                          43
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)
                                                                                          44
       INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                          45
       INTEGER :: ierror
                                                                                          46
in Fortran with the mpi module and mpif.h
                                                                                          47
                                                                                          48
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
```

1	INTEGER IERROR
2	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
3	
4	The free for function is involved to clean up user allocated recourses when the generalized
5	The free_fn function is invoked to clean up user-allocated resources when the generalized
6	request is freed. The free_fn callback is invoked by the MPI_{WAIT TEST}{ANY SOME ALL} call that
7	completed the generalized request associated with this callback. free_fn is invoked after
8	the call to query_fn for the same request. However, if the MPI call completed multiple
9	generalized requests, the order in which free_fn callback functions are invoked is not specified
10	by MPI.
11	The free_fn callback is also invoked for generalized requests that are freed by a call
12	to MPI_REQUEST_FREE (no call to MPI_{WAIT TEST}{ANY SOME ALL} will occur for
13	such a request). In this case, the callback function will be called either in the MPI call
14 15	MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request),
16	whichever happens last, i.e., in this case the actual freeing code is executed as soon as both
17	calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request
18	is not deallocated until after free_fn completes. Note that free_fn will be invoked only once
19	per request by a correct program.
20	
21	Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle
22	to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer
23	valid. However, user copies of this handle are valid until after free_fn completes since
24	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
25	call. Users should note that MPI will deallocate the object after free_fn executes. At
26	this point, user copies of the request handle no longer point to a valid request. MPI will
27	not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid
28	accessing this stale handle. This is a special case in which MPI defers deallocating the
29	object until a later time that is known by the user. (End of advice to users.)
30	
31 32	In C, the cancel function is
33	<pre>typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);</pre>
34	
35	in Fortran with the mpi_f08 module
36	ABSTRACT INTERFACE
37	SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
38	BIND(C)
39	INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
40	LOGICAL :: complete
41	INTEGER :: ierror
42	in Fortran with the mpi module and mpif.h
43	
44	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) INTEGER IERROR
45	INTEGER TERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
46	LOGICAL COMPLETE
47 48	
-10	

The cancel\_fn function is invoked to start the cancelation of a generalized request. It is called by MPI\_CANCEL(request). MPI passes complete=true to the callback function if MPI\_GREQUEST\_COMPLETE was already called on the request, and complete=false otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of 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 1213MPI 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.

query\_fn must not set the error field of status since query\_fn may Advice to users. 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_G	request_complete(MPI_Requ	lest request)
TYPE()	est_complete(request, ier MPI_Request), INTENT(IN) ER, OPTIONAL, INTENT(OUT)	:: request
	EST_COMPLETE(REQUEST, IER ER REQUEST, IERROR	ROR)

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, 41 new nonblocking operations should be defined so that the general semantic rules about MPI 42calls such as MPI\_TEST, MPI\_REQUEST\_FREE, or MPI\_CANCEL still hold. For example, 43 these calls are supposed to be local and nonblocking. Therefore, the callback functions 44query\_fn, free\_fn, or cancel\_fn should invoke blocking MPI communication calls only if the 45context is such that these calls are guaranteed to return in finite time. Once MPI\_CANCEL 46is invoked, the cancelled operation should complete in finite time, irrespective of the state of 47other processes (the operation has acquired "local" semantics). It should either succeed, or 48

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

5

6  $\overline{7}$ 

8 9

10

11

14

15

16

17

18

19

20

21

37

38

39

1 fail without side-effects. The user should guarantee these same properties for newly defined  $\mathbf{2}$ operations. 3 A call to MPI\_GREQUEST\_COMPLETE may unblock a Advice to implementors. 4 blocked user process/thread. The MPI library should ensure that the blocked user 5computation will resume. (End of advice to implementors.) 6 7 12.2.1 Examples 8 9 10 **Example 12.1** This example shows the code for a user-defined reduce operation on an int 11 using a binary tree: each non-root node receives two messages, sums them, and sends them 12up. We assume that no status is returned and that the operation cannot be cancelled. 13 14typedef struct { 15MPI\_Comm comm; 16int tag; 17int root; 18 int valin; 19 int \*valout; 20MPI\_Request request; 21} ARGS; 22 23 $^{24}$ int myreduce(MPI\_Comm comm, int tag, int root, 25int valin, int \*valout, MPI\_Request \*request) 26{ 27ARGS \*args; 28pthread\_t thread; 2930 /\* start request \*/ 31MPI\_Grequest\_start(query\_fn, free\_fn, cancel\_fn, NULL, request); 32 33 args = (ARGS\*)malloc(sizeof(ARGS)); 34 args->comm = comm; 35 args->tag = tag; 36 args->root = root; 37 args->valin = valin; 38 args->valout = valout; 39 args->request = \*request; 40 41 /\* spawn thread to handle request \*/ 42/\* The availability of the pthread\_create call is system dependent \*/ 43 pthread\_create(&thread, NULL, reduce\_thread, args); 44 45return MPI\_SUCCESS; 46} 4748 /\* thread code \*/

```
1
void* reduce_thread(void *ptr)
                                                                                     \mathbf{2}
{
                                                                                     3
   int lchild, rchild, parent, lval, rval, val;
   MPI_Request req[2];
                                                                                     4
   ARGS *args;
                                                                                     5
                                                                                     6
                                                                                     7
   args = (ARGS*)ptr;
                                                                                     8
                                                                                     9
   /* compute left and right child and parent in tree; set
                                                                                     10
      to MPI_PROC_NULL if does not exist */
                                                                                    11
   /* code not shown */
                                                                                    12
   . . .
                                                                                    13
                                                                                    14
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
                                                                                    15
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
                                                                                    16
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    17
   val = lval + args->valin + rval;
   MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
                                                                                    18
                                                                                    19
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
                                                                                    20
   MPI_Grequest_complete((args->request));
                                                                                    21
   free(ptr);
   return(NULL);
                                                                                    22
}
                                                                                    23
                                                                                    24
                                                                                    25
int query_fn(void *extra_state, MPI_Status *status)
                                                                                     26
Ł
   /* always send just one int */
                                                                                    27
   MPI_Status_set_elements(status, MPI_INT, 1);
                                                                                    28
                                                                                    29
   /* can never cancel so always true */
                                                                                    30
   MPI_Status_set_cancelled(status, 0);
                                                                                    31
   /* choose not to return a value for this */
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                    32
                                                                                    33
   /* tag has no meaning for this generalized request */
                                                                                    34
   status->MPI_TAG = MPI_UNDEFINED;
   /* this generalized request never fails */
                                                                                    35
   return MPI_SUCCESS;
                                                                                    36
                                                                                    37
}
                                                                                    38
                                                                                    39
int free_fn(void *extra_state)
                                                                                     40
                                                                                    41
{
                                                                                    42
   /* this generalized request does not need to do any freeing */
   /* as a result it never fails here */
                                                                                    43
                                                                                    44
   return MPI_SUCCESS;
}
                                                                                     45
                                                                                     46
                                                                                     47
                                                                                     48
int cancel_fn(void *extra_state, int complete)
```

L	{	
2		/* This generalized request does not support cancelling.
3		Abort if not already done. If done then treat as if cancel failed.*/
1		if (!complete) {
5		fprintf(stderr,
6		"Cannot cancel generalized request - aborting program\n");
7		MPI_Abort(MPI_COMM_WORLD, 99);
3		}
)		return MPI_SUCCESS;
0	}	
1		

12 13

> 32 33

1

### 12.3 Associating Information with Status

<sup>14</sup> MPI supports several different types of requests besides those for point-to-point operations. <sup>15</sup> These range from MPI calls for I/O to generalized requests. It is desirable to allow these <sup>16</sup> calls to use the same request mechanism, which allows one to wait or test on different <sup>17</sup> types of requests. However, MPI\_{TEST|WAIT}{ANY|SOME|ALL} returns a status with <sup>19</sup> information about the request. With the generalization of requests, one needs to define <sup>19</sup> 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:

```
MPI_STATUS_SET_ELEMENTS(status, datatype, count)
```

```
34
       INOUT
                status
                                            status with which to associate count (Status)
35
       IN
                datatype
                                            datatype associated with count (handle)
36
37
       IN
                count
                                            number of elements to associate with status (integer)
38
39
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
40
                    int count)
41
     MPI_Status_set_elements(status, datatype, count, ierror) BIND(C)
42
         TYPE(MPI_Status), INTENT(INOUT) :: status
43
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
44
         INTEGER, INTENT(IN) :: count
45
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
46
47
     MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
48
```

12.3.	ASS	SOCIATING IN	FORMATION WITH STATUS483	
	INTEG	ER STATUS(MP)	_STATUS_SIZE), DATATYPE, COUNT, IERROR	1 $2$
				3
MPI	STAT	US SET ELEM	ENTS_X(status, datatype, count)	4
	DUT	status	status with which to associate count (Status)	5
	501			6
IN		datatype	datatype associated with count (handle)	7 8
IN		count	number of elements to associate with status (integer)	9
int	MPI_S	tatus_set_ele MPI_Coun	ments_x(MPI_Status *status, MPI_Datatype datatype, ; count)	10 11 12
MPI_	Statu	s_set_element	s_x(status, datatype, count, ierror) BIND(C)	13
			INTENT(INOUT) :: status	14
	TYPE(	[MPI_Datatype]	, INTENT(IN) :: datatype	15
			_COUNT_KIND), INTENT(IN) :: count	16
	INTEG	ER, OPTIONAL	INTENT(OUT) :: ierror	17
MPI_	STATU	S_SET_ELEMENT	S_X(STATUS, DATATYPE, COUNT, IERROR)	18 19
			_STATUS_SIZE), DATATYPE, IERROR	20
	INTEG	ER (KIND=MPI	COUNT_KIND) COUNT	21
	These	functions mod	fy the opaque part of <b>status</b> so that a call to	22
			MPI_GET_ELEMENTS_X will return count. MPI_GET_COUNT	23
		a compatible v		24
				25
			mber of elements is set instead of the count because the former	26
	can o	leal with a non:	ntegral number of datatypes. (End of rationale.)	27 28
	A sub	sequent call to	MPI_GET_COUNT(status, datatype, count) ,	28 29
		-	tus, datatype, count), or MPI_GET_ELEMENTS_X(status, datatype,	30
		•	argument that has the same type signature as the datatype ar-	31
			the call to MPI_STATUS_SET_ELEMENTS or	32
MPI_	STAT	US_SET_ELEN	ENTS_X.	33
	יית		• • • • • • • • • • • • • • • • • • • •	34
			quirement of matching type signatures for these calls is similar at holds when count is set by a receive operation; in that case	35
			at holds when count is set by a receive operation: in that case, 'COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X	36
			with the same signature as the datatype used in the receive call.	37 38
		l of rationale.)		39
		. )		40
				41
MDI	ςтат	US SET CANC	ELLED(status, flag)	42
				43
INC	TUC	status	status with which to associate cancel flag (Status)	44
IN		flag	if true indicates request was cancelled (logical)	45
				46
int	MPI_S	tatus_set_car	celled(MPI_Status *status, int flag)	47
				48

```
    MPI_Status_set_cancelled(status, flag, ierror) BIND(C)
    TYPE(MPI_Status), INTENT(INOUT) :: status
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
    MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
```

INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR

8 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.*)

19 20 21

22 23

 $^{24}$ 

25

26

27

7

9

10

11 12

13 14

15

16 17

18

### 12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, MPI implementations are 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.

31 32

33

38

39

40

41

42

43 44

### 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.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

**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. 

### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

*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 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.)

8 9 10

11

16

17

28

33

34

35

36 37

41

42

43

44

1

2

3

4

5

6

7

**Probe** A receive call that uses source and tag values returned by a preceding call to MPI\_PROBE or MPI\_IPROBE will receive the message matched by the probe call only 12if there was no other matching receive after the probe and before that receive. In a multi-13threaded environment, it is up to the user to enforce this condition using suitable mutual 14exclusion logic. This can be enforced by making sure that each communicator is used by 15only one thread on each process. Alternatively, MPI\_MPROBE or MPI\_IMPROBE can be used.

18 Collective calls Matching of collective calls on a communicator, window, or file handle is 19done according to the order in which the calls are issued at each process. If concurrent 20threads issue such calls on the same communicator, window or file handle, it is up to the 21user to make sure the calls are correctly ordered, using interthread synchronization. 22

- 23Advice to users. With three concurrent threads in each MPI process of a communica- $^{24}$ tor comm, it is allowed that thread A in each MPI process calls a collective operation 25on comm, thread B calls a file operation on an existing filehandle that was formerly 26opened on comm, and thread C invokes one-sided operations on an existing window 27handle that was also formerly created on comm. (End of advice to users.)
- As specified in MPI\_FILE\_OPEN and MPI\_WIN\_CREATE, a file handle Rationale. 29and a window handle inherit only the group of processes of the underlying communi-30 cator, but not the communicator itself. Accesses to communicators, window handles 31and file handles cannot affect one another. (End of rationale.) 32
  - 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 38thread that made the exception-raising MPI call; the exception handler may be executed 39 by a thread that is distinct from the thread that will return the error code. 40
  - The MPI implementation may be multithreaded, so that part of the Rationale. 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)

IN	required	desired level of thread support (integer)	28
- · ·	·		29
OUT	provided	provided level of thread support (integer)	30
			31
int MPI_I	<pre>nit_thread(int *argc, cha</pre>	r *((*argv)[]), int required,	32
	int *provided)		33
MPT Init	thread(required, provided	ierror) BIND(C)	34
	ER, INTENT(IN) :: requir	-	35
	ER, INTENT(OUT) :: provi		36
	ER, OPTIONAL, INTENT(OUT)		37
INIDO			38
MPI_INIT_	THREAD(REQUIRED, PROVIDED	, IERROR)	39
INTEG	ER REQUIRED, PROVIDED, IE	RROR	40
			41

Advice to users. In C, the passing of argc and argv is optional, as with MPI\_INIT as discussed in Section 8.7. In C, null pointers may be passed in their place. (End of advice to users.)

This call initializes MPI in the same way that a call to MPI\_INIT would. In addition, <sup>46</sup> it initializes the thread environment. The argument required is used to specify the desired <sup>47</sup> level of thread support. The possible values are listed in increasing order of thread support. <sup>48</sup>

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

1 **MPI\_THREAD\_SINGLE** Only one thread will execute. 2 **MPI\_THREAD\_FUNNELED** The process may be multi-threaded, but the application must 3 ensure that only the main thread makes MPI calls (for the definition of main thread, 4 see MPI\_IS\_THREAD\_MAIN on page 490). 56 **MPI\_THREAD\_SERIALIZED** The process may be multi-threaded, and multiple threads may 7 make MPI calls, but only one at a time: MPI calls are not made concurrently from 8 two distinct threads (all MPI calls are "serialized"). 9 10 **MPI\_THREAD\_MULTIPLE** Multiple threads may call MPI, with no restrictions. 11These values are monotonic; i.e., MPI\_THREAD\_SINGLE < MPI\_THREAD\_FUNNELED < 12MPI\_THREAD\_SERIALIZED < MPI\_THREAD\_MULTIPLE. 13 Different processes in MPI\_COMM\_WORLD may require different levels of thread sup-14port. 15The call returns in **provided** information about the actual level of thread support that 16will be provided by MPI. It can be one of the four values listed above. 17The level(s) of thread support that can be provided by MPI\_INIT\_THREAD will depend 18 on the implementation, and may depend on information provided by the user before the 19program started to execute (e.g., with arguments to mpiexec). If possible, the call will 20return provided = required. Failing this, the call will return the least supported level such 21that provided > required (thus providing a stronger level of support than required by the 22user). Finally, if the user requirement cannot be satisfied, then the call will return in 23provided the highest supported level. 24A thread compliant MPI implementation will be able to return provided 25= MPI\_THREAD\_MULTIPLE. Such an implementation may always return provided 26= MPI\_THREAD\_MULTIPLE, irrespective of the value of required. 27An MPI library that is not thread compliant must always return 28provided=MPI\_THREAD\_SINGLE, even if MPI\_INIT\_THREAD is called on a multithreaded 29 process. The library should also return correct values for the MPI calls that can be executed 30 before initialization, even if multiple threads have been spawned.  $^{31}$ 32 Such code is erroneous, but if the MPI initialization is performed by a Rationale. 33 library, the error cannot be detected until MPI\_INIT\_THREAD is called. The require-34 ments in the previous paragraph ensure that the error can be properly detected. (End 35 of rationale.) 36 37 A call to MPI\_INIT has the same effect as a call to MPI\_INIT\_THREAD with a required 38= MPI\_THREAD\_SINGLE. 39 Vendors may provide (implementation dependent) means to specify the level(s) of 40thread support available when the MPI program is started, e.g., with arguments to mpiexec. 41 This will affect the outcome of calls to MPI\_INIT and MPI\_INIT\_THREAD. Suppose, for 42example, that an MPI program has been started so that only MPI\_THREAD\_MULTIPLE is 43available. Then  $MPI_INIT_THREAD$  will return provided =  $MPI_THREAD_MULTIPLE$ , irre-44spective of the value of required; a call to MPI\_INIT will also initialize the MPI thread support 45level to MPI\_THREAD\_MULTIPLE. Suppose, instead, that an MPI program has been started 46so that all four levels of thread support are available. Then, a call to MPI\_INIT\_THREAD 47will return provided = required; alternatively, a call to MPI\_INIT will initialize the MPI 48thread support level to MPI\_THREAD\_SINGLE.

*Rationale.* Various optimizations are possible when MPI code is executed singlethreaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits many applications well, e.g., if the process code is a sequential Fortran/C 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. 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.

MPI\_QUERY\_THREAD(provided)

0	UT provided	provided level of thread support (integer)
int	MPI_Query_thread(int *provide	d)
MPI	_Query_thread(provided, ierror INTEGER, INTENT(OUT) :: prov INTEGER, OPTIONAL, INTENT(OUT	ided
MPI	_QUERY_THREAD(PROVIDED, IERROF INTEGER PROVIDED, IERROR	)
	The call returns in <b>provided</b> the cur	rent level of thread support, which will be the value

The call returns in provided the current level of thread support, which will be the value returned in provided by MPI\_INIT\_THREAD, if MPI was initialized by a call to MPI\_INIT\_THREAD().

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

```
1
     MPI_IS_THREAD_MAIN(flag)
2
        OUT
                 flag
                                               true if calling thread is main thread, false otherwise
3
                                               (logical)
4
5
     int MPI_Is_thread_main(int *flag)
6
\overline{7}
     MPI_Is_thread_main(flag, ierror) BIND(C)
8
          LOGICAL, INTENT(OUT) :: flag
9
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
11
          LOGICAL FLAG
12
          INTEGER IERROR
13
14
          This function can be called by a thread to determine if it is the main thread (the thread
15
     that called MPI_INIT or MPI_INIT_THREAD).
16
          All routines listed in this section must be supported by all MPI implementations.
17
18
           Rationale.
                         MPI libraries are required to provide these calls even if they do not
19
           support threads, so that portable code that contains invocations to these functions
20
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
21
           with current MPI codes. (End of rationale.)
22
23
           Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI
24
           call other than MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should
25
           be executed by these threads, until MPI_INIT_THREAD is invoked by one thread
26
           (which, thereby, becomes the main thread). In particular, it is possible to enter the
27
           MPI execution with a multi-threaded process.
28
           The level of thread support provided is a global property of the MPI process that can
29
           be specified only once, when MPI is initialized on that process (or before). Portable
30
           third party libraries have to be written so as to accommodate any provided level of
31
           thread support. Otherwise, their usage will be restricted to specific level(s) of thread
32
           support. If such a library can run only with specific level(s) of thread support, e.g.,
33
           only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check
34
           whether the user initialized MPI to the correct level of thread support and, if not,
35
           raise an exception. (End of advice to users.)
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 [48], collective buffering [7, 15, 49, 53, 59], and disk-directed I/O [44]) 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 8th etype in the file after the displacement. An "explicit offset" is an offset that is used as a formal parameter in explicit data access routines.

### **Unofficial Draft for Comment Only**

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the end of file is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A file pointer is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A file handle is an opaque object created by MPI\_FILE\_OPEN and freed by MPI\_FILE\_CLOSE. All operations on an open file reference the file through the file handle.

#### 13.2 File Manipulation

13.2.1 Opening a File

MPI\_FILE\_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)	21
			22
IN	filename	name of file to open (string)	23
IN	amode	file access mode (integer)	24
IN	info	info object (handle)	25
0.UT	-		26
OUT	fh	new file handle (handle)	27

- int MPI\_File\_open(MPI\_Comm comm, const char \*filename, int amode, MPI\_Info info, MPI\_File \*fh)
- MPI\_File\_open(comm, filename, amode, info, fh, ierror) BIND(C) TYPE(MPI\_Comm), INTENT(IN) :: comm CHARACTER(LEN=\*), INTENT(IN) :: filename INTEGER, INTENT(IN) :: amode TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_File), INTENT(OUT) :: fh INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_FILE\_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
- CHARACTER\*(\*) FILENAME INTEGER COMM, AMODE, INFO, FH, IERROR

42MPI\_FILE\_OPEN opens the file identified by the file name filename on all processes in 43 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 13.7, page 552). A process can

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

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  $\overline{7}$ behavior. 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). 14(End of advice to implementors.) 1516Advice to users. On some implementations of MPI, the file namespace may not be 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.) 22 23Initially, 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, page 500). The constant MPI\_INFO\_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (End of advice to users.)

Files are opened by default using nonatomic mode file consistency semantics (see Section 13.6.1, page 542). 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)

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

```
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)
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
44
     (see Section 13.2.8, page 500). The constant MPI_INFO_NULL refers to the null info, and
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,
48
     whether an open file is deleted or not is also implementation dependent. If the file is not
```

deleted, an error in the class MPI\_ERR\_FILE\_IN\_USE or MPI\_ERR\_ACCESS will be raised. Errors are raised using the default error handler (see Section 13.7, page 552).

Resizing a File 13.2.4 MPI\_FILE\_SET\_SIZE(fh, size) INOUT fh file handle (handle) IN size size to truncate or expand file (integer) int MPI\_File\_set\_size(MPI\_File fh, MPI\_Offset size) MPI\_File\_set\_size(fh, size, ierror) BIND(C) TYPE(MPI\_File), INTENT(IN) :: fh INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_FILE\_SET\_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE

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.

Advice to users. It is possible for the file pointers to point beyond the end of file 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 13.6.1, page 542).

44 45 46

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

34 35

36

37

38 39

40

41

42

```
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
                                               size of the file in bytes (integer)
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
```

INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a data access operation (see Section 13.6.1, page 542).				
13.2.7	Querying File Parameters		7 8	
			9	
MPI_F	ILE_GET_GROUP(fh, group)		10	
IN	fh	file handle (handle)	11 12	
OUT	group	group which opened the file (handle)	13	
int MF	PI_File_get_group(MPI_File :	fh, MPI_Group *group)	14 15	
	le_get_group(fh, group, ie		16 17	
	<pre>YPE(MPI_File), INTENT(IN) : YPE(MPI_G</pre>		18	
	<pre>'PE(MPI_Group), INTENT(OUT) ITEGER, OPTIONAL, INTENT(OU'</pre>	<b>o i</b>	19	
			20	
	LE_GET_GROUP(FH, GROUP, IE TEGER FH, GROUP, IERROR	RRUR)	21 22	
М	PI_FILE_GET_GROUP returns a ne file associated with fh. The gr	duplicate of the group of the communicator used to coup is returned in <b>group</b> . The user is responsible for	23 24 25 26	
			27	
MPI_F	ILE_GET_AMODE(fh, amode)		28	
IN	fh	file handle (handle)	29 30	
OUT	amode	file access mode used to open the file (integer)	31	
			32	
int MF	PI_File_get_amode(MPI_File :	fh, int *amode)	33 34	
	le_get_amode(fh, amode, ie		35	
	<pre>YPE(MPI_File), INTENT(IN) :</pre>		36	
	TEGER, INTENT(OUT) :: amo TEGER, OPTIONAL, INTENT(OU		37	
			38 39	
	LE_GET_AMODE(FH, AMODE, IE TEGER FH, AMODE, IERROR		40	
		in <b>amode</b> , the access mode of the file associated with	41	
fh.		in amore, the access more of the me associated with	42	
			43 44	
		ng an <b>amode</b> bit vector will require a routine such as	45	
0110 1011	the following:			
			47	

```
1
            SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
\mathbf{2}
     i
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 Section 9, page 367) allow a user to provide information such
37
     as file access patterns and file system specifics to direct optimization. Providing hints may
38
     enable an implementation to deliver increased I/O performance or minimize the use of
39
     system resources. However, hints do not change the semantics of any of the I/O interfaces.
40
     In other words, an implementation is free to ignore all hints. Hints are specified on a per
41
     file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and
42
     MPI_FILE_SET_INFO, via the opaque info object. When an info object that specifies a
43
     subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will
44
     be no effect on previously set or defaulted hints that the info does not specify.
45
46
           Advice to implementors. It may happen that a program is coded with hints for one
```

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.*)

```
6
MPI_FILE_SET_INFO(fh, info)
                                                                                              7
            fh
  INOUT
                                         file handle (handle)
                                                                                              8
  IN
            info
                                         info object (handle)
                                                                                              9
                                                                                              10
                                                                                              11
int MPI_File_set_info(MPI_File fh, MPI_Info info)
                                                                                              12
MPI_File_set_info(fh, info, ierror) BIND(C)
                                                                                              13
    TYPE(MPI_File), INTENT(IN) ::
                                         fh
                                                                                              14
    TYPE(MPI_Info), INTENT(IN) ::
                                         info
                                                                                              15
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                              ierror
                                                                                              16
                                                                                              17
MPI_FILE_SET_INFO(FH, INFO, IERROR)
                                                                                              18
    INTEGER FH, INFO, IERROR
                                                                                              19
    MPI_FILE_SET_INFO sets new values for the hints of the file associated with
                                                                                              20
fh. MPI_FILE_SET_INFO is a collective routine. The info object may be different on each
                                                                                             21
process, but any info entries that an implementation requires to be the same on all processes
                                                                                              22
must appear with the same value in each process's info object.
                                                                                              23
                                                                                              ^{24}
     Advice to users. Many info items that an implementation can use when it creates or
                                                                                              25
     opens a file cannot easily be changed once the file has been created or opened. Thus,
                                                                                              26
     an implementation may ignore hints issued in this call that it would have accepted in
                                                                                              27
     an open call. (End of advice to users.)
                                                                                              28
                                                                                              29
                                                                                              30
```

MPI\_FILE\_GET\_INFO(fh, info\_used) IN fh file handle (handle) OUT info\_used new info object (handle) int MPI\_File\_get\_info(MPI\_File fh, MPI\_Info \*info\_used) MPI\_File\_get\_info(fh, info\_used, ierror) BIND(C) TYPE(MPI\_File), INTENT(IN) :: fh TYPE(MPI\_Info), INTENT(OUT) :: info\_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_FILE\_GET\_INFO(FH, INFO\_USED, IERROR) INTEGER FH, INFO\_USED, IERROR

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 pair. The user is responsible for freeing info\_used via MPI\_INFO\_FREE.

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4 5

31

32 33

34 35

36 37

38

39

40 41

42

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, and 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 Section 9, page 367.)

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.
- <sup>35</sup> cb\_buffer\_size (integer) [SAME]: This hint specifies the total buffer space that can be used
   <sup>36</sup> 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.

1

 $\mathbf{2}$ 

3

4

5

6 7

 $^{31}$ 

32

33

34

38

- 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

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)			36 37
INOUT	fh		37 38
		file handle (handle)	39
IN	disp	displacement (integer)	40
IN	etype	elementary datatype (handle)	41
IN	filetype	filetype (handle)	42
IN	datarep	data representation (string)	43
IN	info	info object (handle)	44
		ino object (nanale)	45

### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6

7

8

9 10

11

12

13

14

15 16

17

18

19

20

21 22

23

24 25

26

27

28

29

30

31 32 33

34 35

46

47

1 2 3 4 5 6 7 8	<pre>MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)    TYPE(MPI_File), INTENT(IN) :: fh    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp    TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype    CHARACTER(LEN=*), INTENT(IN) :: datarep    TYPE(MPI_Info), INTENT(IN) :: info    INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)</pre>
9 10 11 12	INTEGER FH, ETYPE, FILETYPE, INFO, IERROR CHARACTER*(*) DATAREP INTEGER(KIND=MPI_OFFSET_KIND) DISP
13 14 15 16 17 18 19 20	The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file. 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. In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.
21 22 23 24 25 26	The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4, page 11), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 13.5.1, page 534 for further details.
27 28 29 30 31	If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.
32 33 34 35 36	Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the <i>displacement</i> may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. ( <i>End of rationale.</i> )
37 38 39 40	Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. ( <i>End of advice to implementors.</i> )
41 42	The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.
43 44 45 46 47	Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.
48	(End of advice to users.)

504

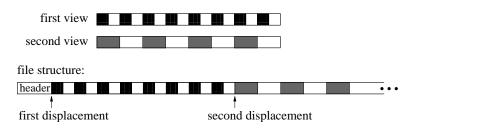


Figure 13.3: Displacements

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 13.5, page 532). (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 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, page 500). 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 13.5, page 532) 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
                                                ierror
         INTEGER, OPTIONAL, INTENT(OUT) ::
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 three <sup>40</sup> 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 types <sup>43</sup> 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 <sup>1</sup>		
		noncollective	collective	2
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL	3
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL	4
	nonblocking &	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN	5
	split collective		MPI_FILE_READ_AT_ALL_END	6
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN	7
			MPI_FILE_WRITE_AT_ALL_END	<u>'</u>
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL	8
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL	9
	nonblocking $\mathcal{E}$	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN	10
	split collective		MPI_FILE_READ_ALL_END	11
		MPI_FILE_IWRITE	MPI_FILE_WRITE_ALL_BEGIN	12
			MPI_FILE_WRITE_ALL_END	13
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED	14
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED	15
	nonblocking $\mathfrak{E}$	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN	
	split collective		MPI_FILE_READ_ORDERED_END	16
		MPI_FILE_IWRITE_SHARED	MPI_FILE_WRITE_ORDERED_BEGIN	<b>J</b> 7
			MPI_FILE_WRITE_ORDERED_END	18
				19

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 13.4.2, page 509.

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 13.4.3, page 513. 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 Section 13.4.4, page 520.

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,

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

### **Unofficial Draft for Comment Only**

 $^{31}$ 

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of predefined datatypes in the typemap of X, and *old\_file\_offset* is the value of the implicit offset before the call. The file position,  $new_file_offset$ , is in terms of a count of etypes relative to the current view.

<sup>6</sup> Synchronism

5

9

- <sup>7</sup>
   MPI supports blocking and nonblocking I/O routines.
  - 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. Given suitable hardware, this allows the transfer of data out of and in to the user's buffer to proceed concurrently with computation. A separate *request complete* call (MPI\_WAIT, MPI\_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. The nonblocking versions of the routines are named MPI\_FILE\_IXXX, where the I stands for immediate.

It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 13.4.5, page 526).

22

33

34

### 23 Coordination

Every noncollective data access routine MPI\_FILE\_XXX has a collective counterpart. For
 most routines, this counterpart is MPI\_FILE\_XXX\_ALL or a pair of MPI\_FILE\_XXX\_BEGIN
 and MPI\_FILE\_XXX\_END. The counterparts to the MPI\_FILE\_XXX\_SHARED routines are
 MPI\_FILE\_XXX\_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4, page 546, for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

## <sup>35</sup><sub>36</sub> Data Access Conventions

<sup>37</sup> Data is moved between files and processes by calling read and write routines. Read routines <sup>38</sup> move data from a file into memory. Write routines move data from memory into a file. The <sup>39</sup> file is designated by a file handle, fh. The location of the file data is specified by an offset <sup>40</sup> into the current view. The data in memory is specified by a triple: buf, count, and datatype. <sup>41</sup> Upon completion, the amount of data accessed by the calling process is returned in a status. <sup>42</sup> An offset designates the starting position in the file for an access. The offset is always in

etype units relative to the current view. Explicit offset routines pass offset as an argument
 (negative values are erroneous). The file pointer routines use implicit offsets maintained by
 MPI.

A data access routine attempts to transfer (read or write) count data items of type
 datatype between the user's buffer buf and the file. The datatype passed to the routine
 must be a committed datatype. The layout of data in memory corresponding to buf, count,

datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 on page 25 and Section 4.1.11 on page 112. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 503). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI\_TEST, MPI\_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20, especially in Sections 17.1.12 and 17.1.13 on pages 628–630 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 17.1.16 to 17.1.19 on pages 633 to 643 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*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).

### 13.4.2 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.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

```
1
     MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)
2
       IN
                 fh
                                             file handle (handle)
3
                 offset
       IN
                                             file offset (integer)
4
5
       OUT
                 buf
                                             initial address of buffer (choice)
6
       IN
                                             number of elements in buffer (integer)
                 count
7
       IN
                 datatype
                                             datatype of each buffer element (handle)
8
9
       OUT
                                             status object (Status)
                 status
10
11
     int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
12
                    MPI_Datatype datatype, MPI_Status *status)
13
     MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
14
          TYPE(MPI_File), INTENT(IN) :: fh
15
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
16
          TYPE(*), DIMENSION(..) :: buf
17
          INTEGER, INTENT(IN) ::
                                     count
18
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
          TYPE(MPI_Status) :: status
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
23
          <type> BUF(*)
24
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
26
         MPI_FILE_READ_AT reads a file beginning at the position specified by offset.
27
28
29
     MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)
30
       IN
                 fh
                                             file handle (handle)
^{31}
32
       IN
                 offset
                                             file offset (integer)
33
       OUT
                 buf
                                             initial address of buffer (choice)
34
       IN
                 count
                                             number of elements in buffer (integer)
35
36
       IN
                                             datatype of each buffer element (handle)
                 datatype
37
       OUT
                 status
                                             status object (Status)
38
39
     int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
40
                    int count, MPI_Datatype datatype, MPI_Status *status)
41
42
     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
43
                    BIND(C)
44
          TYPE(MPI_File), INTENT(IN) :: fh
45
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
46
          TYPE(*), DIMENSION(..) :: buf
47
          INTEGER, INTENT(IN) ::
                                     count
48
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

	(MPI_Status) :: status ER, OPTIONAL, INTENT(OUT	) :: ierror	1 $2$		
MPT FTIF	READ AT ALL (FH OFFSET	RIF COUNT DATATYPE STATUS LERROR)	3		
	<pre>MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>				
• -		STATUS(MPI_STATUS_SIZE), IERROR	5 6		
INTEC	ER(KIND=MPI_OFFSET_KIND)	OFFSET	7		
MPI_I	FILE_READ_AT_ALL is a colle	ective version of the blocking MPI_FILE_READ_AT	8		
interface.		<u> </u>	9		
			10		
MPI_FILE	_WRITE_AT(fh, offset, buf, co	unt, datatype, status)	11 12		
INOUT	fh	file handle (handle)	13		
IN	offset	file offset (integer)	14		
IN	buf	initial address of buffer (choice)	15 16		
IN			17		
	count	number of elements in buffer (integer)	18		
IN	datatype	datatype of each buffer element (handle)	19		
OUT	status	status object (Status)	20		
	'ile envite et/MDT File fb	MDT Offerst offerst courst world which	21 22		
int MPI_F		, MPI_Offset offset, const void *buf, ype datatype, MPI_Status *status)	22		
			24		
		, count, datatype, status, ierror) BIND(C)	25		
	<pre>[MPI_File), INTENT(IN) :: ER(KIND=MPI_OFFSET_KIND)</pre>		26		
	(*), DIMENSION(), INTEN	-	27		
	ER, INTENT(IN) :: count		28 29		
	MPI_Datatype), INTENT(IN	) :: datatype	30		
	(MPI_Status) :: status	· · ·	31		
INTEC	ER, OPTIONAL, INTENT(OUT	) :: ierror	32		
		, COUNT, DATATYPE, STATUS, IERROR)	33		
• -	> BUF(*)		$\frac{34}{35}$		
	ER(KIND=MPI_OFFSET_KIND)	STATUS(MPI_STATUS_SIZE), IERROR	36		
			37		
MPI_I	-ILE_WRITE_AI writes a file	beginning at the position specified by offset.	38		
			39		
			40 41		
			41		
			43		
			44		
	45				
			46 47		
			'± /		

```
1
     MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)
\mathbf{2}
       INOUT
                 fh
                                             file handle (handle)
3
                 offset
       IN
                                             file offset (integer)
4
5
       IN
                 buf
                                             initial address of buffer (choice)
6
       IN
                                             number of elements in buffer (integer)
                 count
7
       IN
                 datatype
                                             datatype of each buffer element (handle)
8
9
       OUT
                                             status object (Status)
                 status
10
11
     int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
12
                     int count, MPI_Datatype datatype, MPI_Status *status)
13
     MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
14
                    BIND(C)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
17
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
18
          INTEGER, INTENT(IN) :: count
19
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                  datatype
20
          TYPE(MPI_Status) :: status
21
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
          <type> BUF(*)
25
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
26
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
27
          MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
28
     MPI_FILE_WRITE_AT interface.
29
30
^{31}
     MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
32
       IN
                 fh
                                             file handle (handle)
33
34
       IN
                 offset
                                             file offset (integer)
35
       OUT
                 buf
                                             initial address of buffer (choice)
36
       IN
                 count
                                             number of elements in buffer (integer)
37
38
       IN
                 datatype
                                             datatype of each buffer element (handle)
39
       OUT
                 request
                                             request object (handle)
40
41
     int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
42
                    MPI_Datatype datatype, MPI_Request *request)
43
44
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
45
                    BIND(C)
46
          TYPE(MPI_File), INTENT(IN) ::
                                             fh
47
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                              offset
48
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
```

```
1
    INTEGER, INTENT(IN) ::
                               count
                                                                                          \mathbf{2}
    TYPE(MPI_Datatype), INTENT(IN) ::
                                            datatype
                                                                                          3
    TYPE(MPI_Request), INTENT(OUT) ::
                                            request
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                                                                          4
                                            ierror
                                                                                          5
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                          6
    <type> BUF(*)
                                                                                          7
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                          8
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
                                                                                          9
                                                                                          10
    MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
                                                                                          11
                                                                                          12
MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
                                                                                          13
                                                                                          14
  INOUT
           fh
                                       file handle (handle)
                                                                                          15
  IN
           offset
                                       file offset (integer)
                                                                                          16
  IN
            buf
                                       initial address of buffer (choice)
                                                                                          17
                                                                                          18
  IN
                                       number of elements in buffer (integer)
           count
                                                                                          19
  IN
           datatype
                                       datatype of each buffer element (handle)
                                                                                          20
  OUT
            request
                                       request object (handle)
                                                                                         21
                                                                                         22
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
                                                                                         23
               int count, MPI_Datatype datatype, MPI_Request *request)
                                                                                          24
                                                                                          25
MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                          26
               BIND(C)
                                                                                          27
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                          28
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                          29
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                          30
    INTEGER, INTENT(IN) :: count
                                                                                          31
    TYPE(MPI_Datatype), INTENT(IN) ::
                                            datatype
                                                                                          32
    TYPE(MPI_Request), INTENT(OUT) ::
                                            request
                                                                                          33
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
                                                                                         34
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                         35
    <type> BUF(*)
                                                                                          36
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                         37
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
                                                                                          38
                                                                                          39
    MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
                                                                                          40
                                                                                          41
13.4.3 Data Access with Individual File Pointers
                                                                                         42
MPI maintains one individual file pointer per process per file handle. The current value
                                                                                          43
                                                                                         44
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.
                                                                                          45
```

The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 509, with the following modification:

#### Unofficial Draft for Comment Only

The shared file pointer is not used nor updated.

46

47

1 2	• the poin		rrent value of the MPI-maintained individual file
3 4 5 6 7 8 9	to point to relative to If MP	the next etype after the last of the current view of the file. I_MODE_SEQUENTIAL mode wa	is initiated, the individual file pointer is updated ne that will be accessed. The file pointer is updated as specified when the file was opened, it is erroneous the exception of MPI_FILE_GET_BYTE_OFFSET.
10 11	MPI_FILE	_READ(fh, buf, count, datatype	e, status)
12	INOUT	fh	file handle (handle)
13	OUT	buf	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	OUT	status	status object (Status)
<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> <li>27</li> <li>28</li> <li>29</li> <li>30</li> <li>31</li> <li>32</li> <li>33</li> <li>34</li> </ol>	MPI_File TYPE TYPE INTEC TYPE INTEC MPI_FILE <type INTEC MPI_</type 	MPI_Status *status) _read(fh, buf, count, data (MPI_File), INTENT(IN) :: (*), DIMENSION() :: bu GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN) (MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) _READ(FH, BUF, COUNT, DATA e> BUF(*) GER FH, COUNT, DATATYPE, S FILE_READ reads a file using t	) :: datatype ) :: ierror ATYPE, STATUS, IERROR) STATUS(MPI_STATUS_SIZE), IERROR the individual file pointer.
35 36	-	<b>13.2</b> The following Fortran of file is reached:	code fragment is an example of reading a file until
37 38 39 40	! Call		until all data has been read. if all requested data is read. nt exits the loop.
41 42 43 44 45 46 47 48	par rea int	rameter (bufsize=100)	

```
1
      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                                                                                       \mathbf{2}
                            MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
                                                                                       3
      call MPI_FILE_SET_VIEW( myfh, zero, MPI_REAL, MPI_REAL, 'native', &
                                                                                       4
                            MPI_INFO_NULL, ierr )
      totprocessed = 0
                                                                                       5
                                                                                       6
      do
                                                                                       7
          call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
                                                                                       8
                                status, ierr )
          call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
                                                                                       9
                                                                                       10
          call process_input( localbuffer, numread )
                                                                                      11
         totprocessed = totprocessed + numread
          if ( numread < bufsize ) exit
                                                                                      12
      enddo
                                                                                      13
                                                                                      14
                                                                                      15
      write(6,1001) numread, bufsize, totprocessed
1001 format( "No more data: read", I3, "and expected", I3, &
                                                                                      16
                                                                                      17
               "Processed total of", I6, "before terminating job." )
                                                                                      18
                                                                                      19
      call MPI_FILE_CLOSE( myfh, ierr )
                                                                                      20
                                                                                      21
                                                                                      22
MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
                                                                                      23
  INOUT
           fh
                                      file handle (handle)
                                                                                      24
                                                                                      25
  OUT
           buf
                                      initial address of buffer (choice)
                                                                                      26
                                      number of elements in buffer (integer)
  IN
           count
                                                                                      27
  IN
           datatype
                                      datatype of each buffer element (handle)
                                                                                      28
                                                                                      29
  OUT
           status
                                      status object (Status)
                                                                                      30
                                                                                      31
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                                                                                      32
              MPI_Datatype datatype, MPI_Status *status)
                                                                                      33
                                                                                      34
MPI_File_read_all(fh, buf, count, datatype, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                      35
    TYPE(*), DIMENSION(..) :: buf
                                                                                      36
                                                                                      37
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                      38
                                                                                      39
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       40
                                                                                      41
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
                                                                                      42
    <type> BUF(*)
                                                                                      43
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                      44
                                                                                      45
    MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
                                                                                       46
                                                                                       47
```

```
1
     MPI_FILE_WRITE(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                 fh
                                             file handle (handle)
3
       IN
                 buf
                                            initial address of buffer (choice)
4
5
                                            number of elements in buffer (integer)
       IN
                 count
6
       IN
                 datatype
                                            datatype of each buffer element (handle)
7
       OUT
                 status
                                            status object (Status)
8
9
     int MPI_File_write(MPI_File fh, const void *buf, int count,
10
11
                    MPI_Datatype datatype, MPI_Status *status)
12
     MPI_File_write(fh, buf, count, datatype, status, ierror) BIND(C)
13
          TYPE(MPI_File), INTENT(IN) :: fh
14
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
15
          INTEGER, INTENT(IN) :: count
16
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
          TYPE(MPI_Status) :: status
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
21
          <type> BUF(*)
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
22
23
          MPI_FILE_WRITE writes a file using the individual file pointer.
24
25
26
     MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
27
       INOUT
                 fh
                                             file handle (handle)
28
       IN
                 buf
                                            initial address of buffer (choice)
29
30
       IN
                 count
                                            number of elements in buffer (integer)
^{31}
       IN
                 datatype
                                            datatype of each buffer element (handle)
32
       OUT
                 status
                                            status object (Status)
33
34
35
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
36
                    MPI_Datatype datatype, MPI_Status *status)
37
     MPI_File_write_all(fh, buf, count, datatype, status, ierror) BIND(C)
38
          TYPE(MPI_File), INTENT(IN) :: fh
39
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
40
          INTEGER, INTENT(IN) :: count
41
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
          TYPE(MPI_Status) :: status
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
46
          <type> BUF(*)
47
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
48
```

MPI\_FILE\_WRITE\_ALL is a collective version of the blocking MPI\_FILE\_WRITE interface.

4 MPI\_FILE\_IREAD(fh, buf, count, datatype, request) 56 INOUT fh file handle (handle) 7 OUT buf initial address of buffer (choice) 8 IN count number of elements in buffer (integer) 9 10 IN datatype datatype of each buffer element (handle) 11 OUT request object (handle) request 1213 int MPI\_File\_iread(MPI\_File fh, void \*buf, int count, 14MPI\_Datatype datatype, MPI\_Request \*request) 1516MPI\_File\_iread(fh, buf, count, datatype, request, ierror) BIND(C) 17TYPE(MPI\_File), INTENT(IN) :: fh 18 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 19 INTEGER, INTENT(IN) :: count 20TYPE(MPI\_Datatype), INTENT(IN) :: datatype 21TYPE(MPI\_Request), INTENT(OUT) :: request 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23MPI\_FILE\_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 24<type> BUF(\*) 25INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 2627MPI\_FILE\_IREAD is a nonblocking version of the MPI\_FILE\_READ interface. 28 **Example 13.3** The following Fortran code fragment illustrates file pointer update seman-2930 tics: 31! Read the first twenty real words in a file into two local 32 buffers. Note that when the first MPI\_FILE\_IREAD returns, ! 33 ! the file pointer has been updated to point to the 34 eleventh real word in the file. Т 35 36 integer bufsize, req1, req2 37 integer, dimension(MPI\_STATUS\_SIZE) :: status1, status2 38 parameter (bufsize=10) 39 buf1(bufsize), buf2(bufsize) real 40 integer (kind=MPI\_OFFSET\_KIND) zero 41 42zero = 043 call MPI\_FILE\_OPEN( MPI\_COMM\_WORLD, 'myoldfile', & 44MPI\_MODE\_RDONLY, MPI\_INFO\_NULL, myfh, ierr ) 45call MPI\_FILE\_SET\_VIEW( myfh, zero, MPI\_REAL, MPI\_REAL, 'native', & 46MPI\_INFO\_NULL, ierr ) 47call MPI\_FILE\_IREAD( myfh, buf1, bufsize, MPI\_REAL, & 48

1 2

```
518
```

```
1
                                    req1, ierr )
\mathbf{2}
            call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
3
                                    req2, ierr )
4
5
            call MPI_WAIT( req1, status1, ierr )
6
            call MPI_WAIT( req2, status2, ierr )
7
8
            call MPI_FILE_CLOSE( myfh, ierr )
9
10
11
     MPI_FILE_IWRITE(fh, buf, count, datatype, request)
12
13
       INOUT
                 fh
                                             file handle (handle)
14
       IN
                 buf
                                            initial address of buffer (choice)
15
       IN
                 count
                                            number of elements in buffer (integer)
16
17
                                            datatype of each buffer element (handle)
       IN
                 datatype
18
       OUT
                 request
                                            request object (handle)
19
20
     int MPI_File_iwrite(MPI_File fh, const void *buf, int count,
21
                    MPI_Datatype datatype, MPI_Request *request)
22
23
     MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)
^{24}
          TYPE(MPI_File), INTENT(IN) :: fh
25
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
26
          INTEGER, INTENT(IN) :: count
27
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
28
          TYPE(MPI_Request), INTENT(OUT) ::
                                                 request
29
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
30
     MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
^{31}
          <type> BUF(*)
32
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
33
34
          MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
35
36
     MPI_FILE_SEEK(fh, offset, whence)
37
38
       INOUT
                 fh
                                             file handle (handle)
39
       IN
                 offset
                                             file offset (integer)
40
       IN
                 whence
                                             update mode (state)
41
42
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
43
44
     MPI_File_seek(fh, offset, whence, ierror) BIND(C)
45
          TYPE(MPI_File), INTENT(IN) :: fh
46
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
47
          INTEGER, INTENT(IN) :: whence
48
```

INT	EGER, OPTIONAI	, INTENT(OUT) :: ierror	1
MPI FIL	E SEEK(FH. OFF	SET, WHENCE, IERROR)	2
	EGER FH, WHENG		3 4
INT	EGER(KIND=MPI	DFFSET_KIND) OFFSET	4 5
MP		ates the individual file pointer according to whence, which has the	6
	g possible values:	ates the individual me pointer according to whence, which has the	7
			8
• MF	$PI_SEEK_SET: the$	pointer is set to offset	9
• MF	$PI_SEEK_CUR: th$	pointer is set to the current pointer position plus offset	10 11
• MF	PI SEEK END: th	pointer is set to the end of file plus offset	12
			13
		ative, which allows seeking backwards. It is erroneous to seek to	14
a negati	ve position in th	view.	15
			16
MPI_FIL	E_GET_POSITIO	N(fh, offset)	17 18
IN	fh	file handle (handle)	19
OUT	offset	offset of individual pointer (integer)	20
001	onset	onset of individual pointer (integer)	21
int MPT	File get nost	tion(MPI_File fh, MPI_Offset *offset)	22
			23
		(fh, offset, ierror) BIND(C)	24
	E(MPI_File), ]		25
		<pre>DFFSET_KIND), INTENT(OUT) :: offset , INTENT(OUT) :: ierror</pre>	26
1111	LGER, UPIIUNAI	, INTENI(UOI) TETIOI	27 28
		(FH, OFFSET, IERROR)	29
	EGER FH, IERRO		30
TNI	EGER(KIND=MP1_	DFFSET_KIND) OFFSET	31
MP	I_FILE_GET_PO	ITION returns, in offset, the current position of the individual file	32
pointer i	n etype units re	tive to the current view.	33
			34
		The offset can be used in a future call to MPI_FILE_SEEK using	35
		_SET to return to the current position. To set the displacement to er position, first convert offset into an absolute byte position using	36
		TE_OFFSET, then call MPI_FILE_SET_VIEW with the resulting	37 38
		of advice to users.)	39
	1		40
			41
	E CET BVTE	FFSET(fh, offset, disp)	42
			43
IN	fh	file handle (handle)	44
IN	offset	offset (integer)	45
OUT	disp	absolute byte position of offset (integer)	46
			47 48

1 2	<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp)</pre>
3 4 5 6 7 8	<pre>MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
9 10 11	MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
12 13 14 15	MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte position. The absolute byte position (from the beginning of the file) of offset relative to the current view of fh is returned in disp.
16 17	13.4.4 Data Access with Shared File Pointers
18 19 20 21 22 23	MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among processes in the communicator group). 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 shared file pointer maintained by MPI. The individual file pointers are not used nor updated.
24 25	The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 509, with the following modifications:
26	• the offset is defined to be the current value of the MPI-maintained shared file pointer,
27 28 29	• the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and
30 31 32	• the use of shared file pointer routines is erroneous unless all processes use the same file view.
33 34 35 36 37	For the noncollective shared file pointer routines, the serialization ordering is not determin- istic. The user needs to use other synchronization means to enforce a specific order. After a shared file pointer operation is initiated, the shared 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.
38 39	
40	
41	
42	
43 44	
45	
46	
47	
48	

Noncollective Operations 1  $\mathbf{2}$ 3 4 MPI\_FILE\_READ\_SHARED(fh, buf, count, datatype, status) 5INOUT fh file handle (handle) 6 OUT buf initial address of buffer (choice) 7 8 IN count number of elements in buffer (integer) 9 IN datatype datatype of each buffer element (handle) 10 OUT status status object (Status) 11 1213 int MPI\_File\_read\_shared(MPI\_File fh, void \*buf, int count, 14MPI\_Datatype datatype, MPI\_Status \*status) 15MPI\_File\_read\_shared(fh, buf, count, datatype, status, ierror) BIND(C) 16TYPE(MPI\_File), INTENT(IN) :: fh 17TYPE(\*), DIMENSION(..) :: buf 18 INTEGER, INTENT(IN) :: count 19 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 20TYPE(MPI\_Status) :: status 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI\_FILE\_READ\_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 24<type> BUF(\*) 25INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR 26MPI\_FILE\_READ\_SHARED reads a file using the shared file pointer. 272829 MPI\_FILE\_WRITE\_SHARED(fh, buf, count, datatype, status) 30 INOUT file handle (handle) fh 3132 IN buf initial address of buffer (choice) 33 IN number of elements in buffer (integer) count 34 IN datatype datatype of each buffer element (handle) 35 36 OUT status status object (Status) 37 38 int MPI\_File\_write\_shared(MPI\_File fh, const void \*buf, int count, 39 MPI\_Datatype datatype, MPI\_Status \*status) 40 MPI\_File\_write\_shared(fh, buf, count, datatype, status, ierror) BIND(C) 41 TYPE(MPI\_File), INTENT(IN) :: fh 42TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 43 INTEGER, INTENT(IN) :: count 44TYPE(MPI\_Datatype), INTENT(IN) :: datatype 45TYPE(MPI\_Status) :: status 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47

```
1
     MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
\mathbf{2}
          <type> BUF(*)
3
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
4
          MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
5
6
\overline{7}
     MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)
8
       INOUT
                 fh
                                             file handle (handle)
9
       OUT
10
                 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
       OUT
14
                 request
                                             request object (handle)
15
16
     int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
17
                    MPI_Datatype datatype, MPI_Request *request)
18
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) BIND(C)
19
          TYPE(MPI_File), INTENT(IN) :: fh
20
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
21
          INTEGER, INTENT(IN) :: count
22
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
          TYPE(MPI_Request), INTENT(OUT) ::
                                                 request
24
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
25
26
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
27
          <type> BUF(*)
28
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
29
          MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
30
     interface.
^{31}
32
33
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
34
       INOUT
                 fh
                                             file handle (handle)
35
36
       IN
                 buf
                                             initial address of buffer (choice)
37
       IN
                                             number of elements in buffer (integer)
                 count
38
                                             datatype of each buffer element (handle)
       IN
                 datatype
39
40
       OUT
                 request
                                             request object (handle)
41
42
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
43
                    MPI_Datatype datatype, MPI_Request *request)
44
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
45
          TYPE(MPI_File), INTENT(IN) :: fh
46
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
47
          INTEGER, INTENT(IN) :: count
48
```

TYPE(MPI_Datatype), INTENT(IN) :: datatype	1
TYPE(MPI_Request), INTENT(OUT) :: request	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
	4
MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	5
<type> BUF(*)</type>	6
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	7
MPI_FILE_IWRITE_SHARED is a nonblocking version of the	8
MPI_FILE_WRITE_SHARED interface.	9
	10
Collective Operations	11
	12
The semantics of a collective access using a shared file pointer is that the accesses to the	13
file will be in the order determined by the ranks of the processes within the group. For each	14
process, the location in the file at which data is accessed is the position at which the shared	15
file pointer would be after all processes whose ranks within the group less than that of this	16
process had accessed their data. In addition, in order to prevent subsequent shared offset	17
accesses by the same processes from interfering with this collective access, the call might	18
return only after all the processes within the group have initiated their accesses. When the	19
call returns, the shared file pointer points to the next etype accessible, according to the file	20
view used by all processes, after the last etype requested.	21
	22

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.*)

#### MPI\_FILE\_READ\_ORDERED(fh, buf, count, datatype, status)

			-,	
	INOUT	fh	file handle (handle)	37
				38
	OUT	buf	initial address of buffer (choice)	39
	IN	count	number of elements in buffer (integer)	40
	IN	datatype	datatype of each buffer element (handle)	41
		5.		42
	OUT	status	status object (Status)	43
				44
i	nt MPI_Fi	ile_read_ordered(MPI_File	fh, void *buf, int count,	45
		MPI_Datatype datatype	e, MPI_Status *status)	46
м	PT File 1	read ordered (fh buf cour	nt, datatype, status, ierror) BIND(C)	47
1.		.cuu_orucrcu(III, Dur, Cou	nt, addatype, status, ierior, bind(c)	48

23

 $^{24}$ 

25

26

27

28 29

30

 $^{31}$ 

32

33 34 35

```
524
```

```
1
         TYPE(MPI_File), INTENT(IN) ::
                                           fh
\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_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
8
         <type> BUF(*)
9
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
10
11
         MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED
12
     interface.
13
14
     MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
15
16
       INOUT
                fh
                                           file handle (handle)
17
                buf
                                           initial address of buffer (choice)
       IN
18
       IN
                                           number of elements in buffer (integer)
                count
19
20
       IN
                datatype
                                           datatype of each buffer element (handle)
21
       OUT
                status
                                           status object (Status)
22
23
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
24
                    MPI_Datatype datatype, MPI_Status *status)
25
26
     MPI_File_write_ordered(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_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
         <type> BUF(*)
35
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
36
37
         MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
38
     interface.
39
40
     Seek
41
     If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
42
     to call the following two routines (MPI_FILE_SEEK_SHARED and
43
     MPI_FILE_GET_POSITION_SHARED).
44
45
46
47
48
```

MPI_FILE	_SEEK_SHARED(fh, offset, wh	ience)	1		
INOUT	fh	file handle (handle)	2		
IN	offset	file offset (integer)	3 4		
IN	whence	update mode (state)	5		
			6		
int MPI_F	Sile_seek_shared(MPI_File	fh, MPI_Offset offset, int whence)	7		
	seek_shared(fh, offset, )		8		
	[MPI_File), INTENT(IN) ::	fh	9 10		
	ER(KIND=MPI_OFFSET_KIND)		10		
	ER, INTENT(IN) :: whence		12		
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	13		
MPI_FILE_	SEEK_SHARED(FH, OFFSET, N	WHENCE, IERROR)	14		
	ER FH, WHENCE, IERROR		15		
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	16 17		
MPI_I	FILE_SEEK_SHARED updates	the shared file pointer according to whence, which	18		
	lowing possible values:		19		
<ul> <li>MDI</li> </ul>	SEEK_SET: the pointer is set t	to offect	20		
	SEEK_SET. the pointer is set	to onset	21		
<ul> <li>MPI_</li> </ul>	SEEK_CUR: the pointer is set	to the current pointer position plus offset	22		
MPI	SEEK_END: the pointer is set	to the end of file plus offset	23 24		
	-	-	25		
		tive; all the processes in the communicator group	26		
	nd whence.	all MPI_FILE_SEEK_SHARED with the same values	27		
		llows seeking backwards. It is erroneous to seek to	28		
	position in the view.		29		
	-		30 31		
	_GET_POSITION_SHARED(fh	offcot)	32		
	· · · · · · · · · · · · · · · · · · ·	,	33		
IN	fh	file handle (handle)	34		
OUT	offset	offset of shared pointer (integer)	35		
			36 37		
int MPI_F	'ile_get_position_shared(	MPI_File fh, MPI_Offset *offset)	38		
	get_position_shared(fh, o		39		
	<pre>(MPI_File), INTENT(IN) ::</pre>		40		
	ER(KIND=MPI_OFFSET_KIND) ER, OPTIONAL, INTENT(OUT)		41		
			42		
	GET_POSITION_SHARED(FH, (	OFFSET, IERROR)	43 44		
	ER FH, IERROR ER(KIND=MPI_OFFSET_KIND)	OFFSFT	45		
			46		
	MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the 47				
shared file pointer in etype units relative to the current view. $^{48}$					

Advice to users. The offset can be used in a future call to MPI\_FILE\_SEEK\_SHARED using whence = MPI\_SEEK\_SET to return to the current position. To set the displacement to the current file pointer position, first convert offset into an absolute byte position using MPI\_FILE\_GET\_BYTE\_OFFSET, then call MPI\_FILE\_SET\_VIEW with the resulting displacement. (*End of advice to users.*)

13.4.5 Split Collective Data Access Routines

MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-cesses 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. 

<sup>17</sup> Split collective data access operations on a file handle fh are subject to the semantic <sup>18</sup> rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI\_FILE\_READ\_ALL\_BEGIN) or the end call (e.g., MPI\_FILE\_READ\_ALL\_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI\_FILE\_READ\_ALL on one process does not match an MPI\_FILE\_READ\_ALL\_BEGIN/ MPI\_FILE\_READ\_ALL\_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 17.1.17 on page 634, but not all of the problems described in Section 17.1.16 on page 633.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

 $^{24}$ 

1 MPI\_File\_read\_all\_begin(fh, ...);  $\mathbf{2}$ . . . 3 MPI\_File\_read\_all(fh, ...); 4 . . . MPI\_File\_read\_all\_end(fh, ...); 5 6 is erroneous. 7 8 • In a multithreaded implementation, any split collective begin and end operation called 9 by a process must be called from the same thread. This restriction is made to simplify 10 the implementation in the multithreaded case. (Note that we have already disallowed 11 having two threads begin a split collective operation on the same file handle since only 12one split collective operation can be active on a file handle at any time.) 13 14The arguments for these routines have the same meaning as for the equivalent collective 15versions (e.g., the argument definitions for MPI\_FILE\_READ\_ALL\_BEGIN and 16MPI\_FILE\_READ\_ALL\_END are equivalent to the arguments for MPI\_FILE\_READ\_ALL). 17 The begin routine (e.g., MPI\_FILE\_READ\_ALL\_BEGIN) begins a split collective operation 18 that, when completed with the matching end routine (i.e., MPI\_FILE\_READ\_ALL\_END) 19 produces the result as defined for the equivalent collective routine (i.e., 20MPI\_FILE\_READ\_ALL). 21For the purpose of consistency semantics (Section 13.6.1, page 542), a matched pair 22 of split collective data access operations (e.g., MPI\_FILE\_READ\_ALL\_BEGIN and 23MPI\_FILE\_READ\_ALL\_END) compose a single data access.  $^{24}$ 2526MPI\_FILE\_READ\_AT\_ALL\_BEGIN(fh, offset, buf, count, datatype) 27IN fh file handle (handle) 28 IN offset file offset (integer) 29 30 OUT buf initial address of buffer (choice) 31IN count number of elements in buffer (integer) 32 IN datatype datatype of each buffer element (handle) 33 34 int MPI\_File\_read\_at\_all\_begin(MPI\_File fh, MPI\_Offset offset, void \*buf, 35 int count, MPI\_Datatype datatype) 36 37 MPI\_File\_read\_at\_all\_begin(fh, offset, buf, count, datatype, ierror) 38 BIND(C) 39 TYPE(MPI\_File), INTENT(IN) :: fh 40 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 41 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 42INTEGER, INTENT(IN) :: count 43 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45MPI\_FILE\_READ\_AT\_ALL\_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 4647<type> BUF(\*) 48 INTEGER FH, COUNT, DATATYPE, IERROR

```
1
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
\mathbf{2}
3
4
     MPI_FILE_READ_AT_ALL_END(fh, buf, status)
5
       IN
                 fh
                                            file handle (handle)
6
       OUT
7
                 buf
                                            initial address of buffer (choice)
8
       OUT
                                            status object (Status)
                status
9
10
     int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
11
12
     MPI_File_read_at_all_end(fh, buf, status, ierror) BIND(C)
         TYPE(MPI_File), INTENT(IN) :: fh
13
14
         TYPE(*), DIMENSION(...), ASYNCHRONOUS ::
                                                       buf
15
         TYPE(MPI_Status) :: status
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
18
          <type> BUF(*)
19
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
20
21
22
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
23
^{24}
       INOUT
                fh
                                            file handle (handle)
25
       IN
                offset
                                            file offset (integer)
26
       IN
                buf
                                            initial address of buffer (choice)
27
28
       IN
                count
                                            number of elements in buffer (integer)
29
       IN
                datatype
                                            datatype of each buffer element (handle)
30
^{31}
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, const
32
                    void *buf, int count, MPI_Datatype datatype)
33
34
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
35
                    BIND(C)
36
         TYPE(MPI_File), INTENT(IN) ::
                                            fh
37
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
38
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
39
         INTEGER, INTENT(IN) :: count
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
43
         <type> BUF(*)
44
         INTEGER FH, COUNT, DATATYPE, IERROR
45
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
46
47
48
```

1 MPI\_FILE\_WRITE\_AT\_ALL\_END(fh, buf, status)  $\mathbf{2}$ INOUT fh file handle (handle) 3 IN buf initial address of buffer (choice) 4 OUT 5status status object (Status) 6 7 int MPI\_File\_write\_at\_all\_end(MPI\_File fh, const void \*buf, 8 MPI\_Status \*status) 9 MPI\_File\_write\_at\_all\_end(fh, buf, status, ierror) BIND(C) 10 TYPE(MPI\_File), INTENT(IN) :: fh 11 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 12TYPE(MPI\_Status) :: status 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_FILE\_WRITE\_AT\_ALL\_END(FH, BUF, STATUS, IERROR) 16<type> BUF(\*) 17INTEGER FH, STATUS(MPI\_STATUS\_SIZE), IERROR 18 19 20MPI\_FILE\_READ\_ALL\_BEGIN(fh, buf, count, datatype) 21INOUT fh file handle (handle) 22 23OUT buf initial address of buffer (choice)  $^{24}$ IN count number of elements in buffer (integer) 25IN datatype datatype of each buffer element (handle) 262728 int MPI\_File\_read\_all\_begin(MPI\_File fh, void \*buf, int count, 29 MPI\_Datatype datatype) 30 MPI\_File\_read\_all\_begin(fh, buf, count, datatype, ierror) BIND(C) 31TYPE(MPI\_File), INTENT(IN) :: fh 32 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 33 INTEGER, INTENT(IN) :: count 34 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_FILE\_READ\_ALL\_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) 38 <type> BUF(\*) 39 INTEGER FH, COUNT, DATATYPE, IERROR 40 41 42MPI\_FILE\_READ\_ALL\_END(fh, buf, status) 43 INOUT fh file handle (handle) 44OUT 45buf initial address of buffer (choice) 46OUT status object (Status) status 47

```
1
     int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
\mathbf{2}
     MPI_File_read_all_end(fh, buf, status, ierror) BIND(C)
3
         TYPE(MPI_File), INTENT(IN) :: fh
4
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
5
         TYPE(MPI_Status) :: status
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
9
         <type> BUF(*)
10
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
11
12
13
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
14
       INOUT
                fh
                                           file handle (handle)
15
16
       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
20
21
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype)
22
23
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror) BIND(C)
24
         TYPE(MPI_File), INTENT(IN) :: fh
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_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
30
^{31}
         <type> BUF(*)
         INTEGER FH, COUNT, DATATYPE, IERROR
32
33
34
35
     MPI_FILE_WRITE_ALL_END(fh, buf, status)
36
       INOUT
                fh
                                           file handle (handle)
37
       IN
                buf
                                           initial address of buffer (choice)
38
39
       OUT
                status
                                           status object (Status)
40
41
     int MPI_File_write_all_end(MPI_File fh, const void *buf,
42
                    MPI_Status *status)
43
     MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)
44
         TYPE(MPI_File), INTENT(IN) :: fh
45
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::
46
                                                                   buf
47
         TYPE(MPI_Status) :: status
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

	WRITE_ALL_END(FH, BUF, ST > BUF(*)	CATUS, IERROR)	1 2
• 1	ER FH, STATUS(MPI_STATUS_	SIZE), IERROR	3
			4
			5
MPI_FILE_	READ_ORDERED_BEGIN(fh,	buf, count, datatype)	6
INOUT	fh	file handle (handle)	7 8
OUT	buf	initial address of buffer (choice)	9
IN	count	number of elements in buffer (integer)	10
IN	datatype	datatype of each buffer element (handle)	11
	udiatype	datatype of each build cicilian (nanole)	12 13
int MPI_F	ile_read_ordered_begin(MF MPI_Datatype datatype	PI_File fh, void *buf, int count,	14
			15 16
	-	if, count, datatype, ierror) BIND(C)	17
	<pre>MPI_File), INTENT(IN) :: *), DIMENSION(), ASYNCH</pre>		18
	ER, INTENT(IN) :: count		19
	MPI_Datatype), INTENT(IN)	:: datatype	20
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	21 22
MPI FILE	READ ORDERED BEGIN(FH. BU	JF, COUNT, DATATYPE, IERROR)	22
	> BUF(*)		24
INTEG	ER FH, COUNT, DATATYPE, J	ERROR	25
			26
			27
MPI_FILE_	READ_ORDERED_END(fh, b	uf, status)	28 29
INOUT	fh	file handle (handle)	30
OUT	buf	initial address of buffer (choice)	31
OUT	status	status object (Status)	32
			33
int MPI_F	ile_read_ordered_end(MPI_	File fh, void *buf, MPI_Status *status)	34
MPI File	read_ordered_end(fh, buf,	status, ierror) BIND(C)	35 36
	<pre>MPI_File), INTENT(IN) ::</pre>		37
TYPE(	*), DIMENSION(), ASYNCH	IRONOUS :: buf	38
	MPI_Status) :: status		39
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	40
MPI_FILE_	READ_ORDERED_END(FH, BUF,	STATUS, IERROR)	41
	> BUF(*)		42 43
INTEG	ER FH, STATUS(MPI_STATUS_	SIZE), IERROR	43 44
			45
			46
			47
			48

```
1
     MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
2
       INOUT
                fh
                                            file handle (handle)
3
                buf
       IN
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
                datatype
       IN
                                            datatype of each buffer element (handle)
7
8
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
9
                    MPI_Datatype datatype)
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_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
18
          <type> BUF(*)
19
         INTEGER FH, COUNT, DATATYPE, IERROR
20
21
22
     MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
23
24
       INOUT
                fh
                                            file handle (handle)
25
       IN
                buf
                                            initial address of buffer (choice)
26
       OUT
                status
                                            status object (Status)
27
28
29
     int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
30
                    MPI_Status *status)
^{31}
     MPI_File_write_ordered_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_ORDERED_END(FH, BUF, STATUS, IERROR)
38
          <type> BUF(*)
39
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
40
41
42
     13.5
             File Interoperability
43
```

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 13.5.2, page 536) as well as the data conversion functions (Section 13.5.3, page 537).

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 13.6.1, page 542), 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.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility that transfers and converts files between the native file system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. In a high-quality implementation of MPI, users will be able to manipulate MPI files using the same or similar tools that the native file system offers for manipulating its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports multiple data representations: "native," "internal," and "external32." An implementation may support additional data representations. MPI also supports userdefined data representations (see Section 13.5.3, page 537). 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
 the loss of transparent interoperability within a heterogeneous MPI environment.

#### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11 12

13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

1	Advice to users. This data representation should only be used in a homogeneous
2	MPI environment, or when the MPI application is capable of performing the data
3	type conversions itself. (End of advice to users.)
4	
5	Advice to implementors. When implementing read and write operations on
6	top of MPI message-passing, the message data should be typed as MPI_BYTE
7	to ensure that the message routines do not perform any type conversions on the
8	data. (End of advice to implementors.)
9	
10	"internal" This data representation can be used for I/O operations in a homogeneous
11	or heterogeneous environment; the implementation will perform type conversions if
12	necessary. The implementation is free to store data in any format of its choice, with
13	the restriction that it will maintain constant extents for all predefined datatypes in any
14	one file. The environment in which the resulting file can be reused is implementation-
15	defined and must be documented by the implementation.
16	
17	Rationale. This data representation allows the implementation to perform $I/O$
18	efficiently in a heterogeneous environment, though with implementation-defined
19	restrictions on how the file can be reused. (End of rationale.)
20	
21	Advice to implementors. Since "external32" is a superset of the functionality
22	provided by "internal," an implementation may choose to implement "internal"
23	as "external32." (End of advice to implementors.)
24	"external 22" This data representation states that read and write experience convert all
25	"external32" This data representation states that read and write operations convert all
26	data from and to the "external32" representation defined in Section 13.5.2, page 536.
27	The data conversion rules for communication also apply to these conversions (see
28	Section 3.3.2, page 25-27, of the MPI-1 document). The data on the storage medium
29	is always in this canonical representation, and the data in memory is always in the
30	local process's native representation.
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	their respective native representations. Second, the file can be exported from one MPI
34	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	The disadvantage of this data representation is that data precision and I/O perfor-
37	mance may be lost in data type conversions.
38	
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	13.5.1 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, 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 11 of a file server. The etype and filetype are interpreted as if they were defined at this 12file server, by the same sequence of calls used to define them at the calling process. 13 If the data representation is "native", then this logical file server runs on the same 14architecture as the calling process, so that these types define the same data layout 15on the file as they would define in the memory of the calling process. If the etype 16and filetype are portable datatypes, then the data layout defined in the file is the 17 same as would be defined in the calling process memory, up to a scaling factor. The 18 routine MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate this scaling factor. 19Thus, two equivalent, portable datatypes will define the same data layout in the file, 20even in a heterogeneous environment with "internal", "external32", or user defined 21data representations. Otherwise, the etype and filetype must be constructed so that 22their typemap and extent are the same on any architecture. This can be achieved 23if 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, 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.*)

- 43 44 45
- $46 \\ 47$

48

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10

25

26

27

28

29

30 31

32

33

34

35

36

37

38

39

40

41

```
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 13.5.3, page 537), 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
24
           of a derived datatype can be calculated by first determining the extents of the prede-
25
           fined datatypes in this derived datatype using dtype_file_extent_fn (see Section 13.5.3,
26
           page 537). (End of advice to implementors.)
27
28
     13.5.2 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
      0 implies false and nonzero implies true. C float _Complex, double _Complex and long
40
      double _Complex as well as Fortran COMPLEX and DOUBLE COMPLEX are represented by a
41
      pair of floating point format values for the real and imaginary components. Characters are
42
      in ISO 8859-1 format [38]. Wide characters (of type MPI_WCHAR) are in Unicode format
43
     [60].
44
          All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant
45
      bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary
```

<sup>46</sup> 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."

Advice to implementors. The MPI treatment of "NaN" is similar to the approach used	1
in XDR (see ftp://ds.internic.net/rfc/rfc1832.txt). (End of advice to implementors.)	2
	3
All data is byte aligned, regardless of type. All data items are stored contiguously in	4
the file (if the file view is contiguous).	5
	6
Advice to implementors. All bytes of LOGICAL and bool must be checked to determine	7
the value. (End of advice to implementors.)	8
	9
Advice to users. The type MPI_PACKED is treated as bytes and is not converted.	10
The user should be aware that MPI_PACK has the option of placing a header in the	11
beginning of the pack buffer. (End of advice to users.)	12
The size of the predefined datatypes returned from MPI TYPE CREATE F90 REAL.	13

The size of the predefined datatypes returned from MPI\_TYPE\_CREATE\_F90\_REAL, MPI\_TYPE\_CREATE\_F90\_COMPLEX, and MPI\_TYPE\_CREATE\_F90\_INTEGER are defined in Section 17.1.9, page 621.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (End of advice to implementors.)

Table 13.2 specifies the sizes of predefined datatypes in "external32" format.

#### 13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and

2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

# MPI\_REGISTER\_DATAREP(datarep, read\_conversion\_fn, write\_conversion\_fn, dtype\_file\_extent\_fn, extra\_state)

			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 to file representation (function)	40 41
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)	42 43 44
IN	extra_state	extra state	45 46

int MPI\_Register\_datarep(const char \*datarep,

MPI\_Datarep\_conversion\_function \*read\_conversion\_fn,

#### Unofficial Draft for Comment Only

 $^{31}$ 

Туре	Length	Optional Type	Length
MPI_PACKED	1	MPI_INTEGER1	1
MPI_BYTE	1	MPI_INTEGER2	2
MPI_CHAR	1	MPI_INTEGER4	4
MPI_UNSIGNED_CHAR	1	MPI_INTEGER8	8
MPI_SIGNED_CHAR	1	MPI_INTEGER16	16
MPI_WCHAR	2		
MPI_SHORT	2	MPI_REAL2	2
MPI_UNSIGNED_SHORT	2	MPI_REAL4	4
MPI_INT	4	MPI_REAL8	8
MPI_UNSIGNED	4	MPI_REAL16	16
MPI_LONG	4		
MPI_UNSIGNED_LONG	4	MPI_COMPLEX4	2*2
MPI_LONG_LONG_INT	8	MPI_COMPLEX8	2*4
MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16	2*8
MPI_FLOAT	4	MPI_COMPLEX32	2*16
MPI_DOUBLE	8	_	
1PI_LONG_DOUBLE	16		
MPI_C_BOOL	1		
MPI_INT8_T	1	C++ Types	Length
MPI_INT16_T	2		
MPI_INT32_T	4	MPI_CXX_BOOL	1
MPI_INT64_T	8	MPI_CXX_FLOAT_COMPLEX	2*4
MPI_UINT8_T	1	MPI_CXX_DOUBLE_COMPLEX	2*8
MPI_UINT16_T	2	MPI_CXX_LONG_DOUBLE_COM	PLEX 2*16
MPI_UINT32_T	4		
MPI_UINT64_T	8		
MPI_AINT	8		
MPI_COUNT	8		
MPI_OFFSET	8		
MPI_C_COMPLEX	2*4		
MPI_C_FLOAT_COMPLEX	2*4		
MPI_C_DOUBLE_COMPLEX	2*8		
MPI_C_LONG_DOUBLE_COMPLEX			
MPI_CHARACTER	1		
MPI_LOGICAL	4		
MPI_INTEGER	4		
MPI_REAL	4		
MPI_DOUBLE_PRECISION	8		
MPI_COMPLEX	2*4		
MPI_DOUBLE_COMPLEX	2*8		
	2.0		
Table 13.2:	"external32"	sizes of predefined datatypes	

Extent Callback

```
1
              MPI_Datarep_conversion_function *write_conversion_fn,
                                                                                        2
              MPI_Datarep_extent_function *dtype_file_extent_fn,
                                                                                        3
              void *extra_state)
                                                                                        4
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
                                                                                        5
              dtype_file_extent_fn, extra_state, ierror) BIND(C)
                                                                                        6
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                        7
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
    PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
                                                                                        9
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
                                                                                        10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                        11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        12
                                                                                        13
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
              DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
                                                                                        14
                                                                                        15
    CHARACTER*(*) DATAREP
                                                                                        16
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
                                                                                        17
    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-
                                                                                        26
tion 13.7, page 552). 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.
                                                                                       28
No routines are provided to delete data representations and free the associated resources;
                                                                                       29
it is not expected that an application will generate them in significant numbers.
                                                                                        30
                                                                                        31
```

```
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

The function dtype\_file\_extent\_fn must return, in file\_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call. MPI will only call 48

#### Unofficial Draft for Comment Only

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 native
25
     representation. Before calling this routine, MPI allocates and fills filebuf with
26
     count contiguous data items. The type of each data item matches the corresponding entry
27
     for the predefined datatype in the type signature of datatype. The function is passed, in
28
     extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The
29
     function must copy all count data items from filebuf to userbuf in the distribution described
30
     by datatype, converting each data item from file representation to native representation.
^{31}
     datatype will be equivalent to the datatype that the user passed to the read function. If the
32
     size of datatype is less than the size of the count data items, the conversion function must
33
     treat datatype as being contiguously tiled over the userbuf. The conversion function must
34
     begin storing converted data at the location in userbuf specified by position into the (tiled)
35
     datatype.
36
37
           Advice to users. Although the conversion functions have similarities to MPI_PACK
38
           and MPI_UNPACK, one should note the differences in the use of the arguments count
39
           and position. In the conversion functions, count is a count of data items (i.e., count
40
           of typemap entries of datatype), and position is an index into this typemap. In
41
           MPI_PACK, incount refers to the number of whole datatypes, and position is a number
42
           of bytes. (End of advice to users.)
43
           Advice to implementors. A converted read operation could be implemented as follows:
44
45
             1. Get file extent of all data items
46
             2. Allocate a filebuf large enough to hold all count data items
47
48
             3. Read data from file into filebuf
```

- 4. Call read\_conversion\_fn to convert data and place it into userbuf
- 5. Deallocate filebuf

#### (End of advice to implementors.)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

*Rationale.* Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write\_conversion\_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the 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 47 (read\_conversion\_fn, write\_conversion\_fn, and dtype\_file\_extent\_fn) when one of the read or 48

#### Unofficial Draft for Comment Only

1

2

3 4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15 16

17

18

19 20

21

22

23

 $^{24}$ 

25 26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

write routines in Section 13.4, page 506, 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

23

24

25

26 27

28

29

30

31

32 33

34

35 36

37

38

39 40 41

42

8

9

# 13.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file. In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

<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.

- The data access routines directly use types enumerated in Section 13.5.2, page 536, 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, page 617).
  - For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatiblity with another implementation's "native" or "internal" representation.

Advice to users. Section 17.1.9, page 617, defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

# 13.6 Consistency and Semantics

<sup>43</sup> 13.6.1 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 13.4.5, page 526) 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.

#### Unofficial Draft for Comment Only

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 13.6.10, page 548, 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	IN	flag	true to set atomic mode, false to set nonatomic mode	
21 22		5	(logical)	
23				
24	int MPI_F	'ile_set_atomicity(MPI_Fi	le fh, int flag)	
25	MPI_File_	set_atomicity(fh, flag, :	ierror) BIND(C)	
26	TYPE(MPI_File), INTENT(IN) :: fh			
27 28	LOGIC	AL, INTENT(IN) :: flag		
28 29	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	
30	MPI_FILE_	SET_ATOMICITY(FH, FLAG, 1	IERROR)	
31	INTEG	ER FH, IERROR		
32	LOGIC	AL FLAG		
33	тип			

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

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

14

15

16 17

MPI_FILE_GET_ATOMICITY(fh, flag) <sup>1</sup>						
IN fh	file handle (handle)	2				
OUT flag	true if atomic mode, false if nonatomic mode (logical)	3				
	the if atomic more, tase if ionatomic more (logical)	4 5				
int MDI Filo got stomicity(MDI Fi	lo fh int *flog)	6				
<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>						
<pre>MPI_File_get_atomicity(fh, flag, ierror) BIND(C)</pre>						
TYPE(MPI_File), INTENT(IN) :: fh						
LOGICAL, INTENT(OUT) :: flag		10				
INTEGER, OPTIONAL, INTENT(OUT)	) :: ierror	11				
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR						
					LOGICAL FLAG	
MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access						
operations on the set of file handles created by one collective open. If flag is true, atomic						
mode is enabled; if flag is false, nonatomic mode is enabled.						
		18				
		19 20				
MPI_FILE_SYNC(fh)		20 21				
INOUT fh	file handle (handle)	22				
		23				
<pre>int MPI_File_sync(MPI_File fh)</pre>		24				
MDT Eile and (the issues) DIND(Q)						
<pre>MPI_File_sync(fh, ierror) BIND(C)     TYPE(MPI_File), INTENT(IN) ::</pre>	fh	26				
INTEGER, OPTIONAL, INTENT(OUT		27				
	,	28				
MPI_FILE_SYNC(FH, IERROR)		29				
INTEGER FH, IERROR		30 31				
Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process to be transferred to the storage device. If other processes have made updates to the storage						
					device, then all such updates become visible to subsequent reads of <b>fh</b> by the calling process.	
MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see						
above). MPI_FILE_SYNC is a collective operation.						
				-	The user is responsible for ensuring that all nonblocking requests and split collective	
operations on fh have been completed before calling MPI_FILE_SYNC—otherwise, the call to MPI_FILE_SYNC is erroneous.						
to with the the is entoneous.		40				
13.6.2 Random Access vs. Sequential	Files	41				
13.0.2 Nandom Access VS. Sequential Thes						
MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and						
					re erroneous, and the pointer update rules specified	47 48
				WITTLE_GET_FOSTION_SHARED at	e enoncous, and the pointer update fulles specified	

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.

13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

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

# 13.6.4 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 dup of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

13.6.5 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI\_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI\_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

# Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11 12

### 13.6.6 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI\_FILE\_OPEN, or the etype and filetype used in an MPI\_FILE\_SET\_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI\_FILE\_SET\_VIEW, and the datatype must be committed before calling MPI\_FILE\_READ or MPI\_FILE\_WRITE.

# 13.6.7 MPI\_Offset Type

MPI\_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI\_Offset.

In Fortran, the corresponding integer is an integer with kind parameter MPI\_OFFSET\_KIND, which is defined in the mpi\_f08 module, the mpi module and the mpif.h include file.

In Fortran 77 environments that do not support KIND parameters, MPI\_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI\_Offset are similar to those for addresses (see Section 17.2, page 647).

### 13.6.8 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program 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, page 500).

### 13.6.9 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI\_FILE\_SET\_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI\_FILE\_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI\_FILE\_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI\_FILE\_OPEN, returned.

When applying consistency semantics, calls to MPI\_FILE\_SET\_SIZE and MPI\_FILE\_PREALLOCATE are considered writes to the file (which conflict with operations

### Unofficial Draft for Comment Only

 $^{31}$ 

 $45 \\ 46$ 

```
that access bytes at displacements between the old and new file sizes), and
\mathbf{2}
      MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the
3
      file).
4
           Advice to users. Any sequence of operations containing the collective routines
5
           MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such,
6
           sequential consistency in nonatomic mode is not guaranteed unless the conditions in
7
           Section 13.6.1, page 542, are satisfied. (End of advice to users.)
8
9
          File pointer update semantics (i.e., file pointers are updated by the amount accessed)
10
      are only guaranteed if file size changes are sequentially consistent.
11
12
           Advice to users.
                              Consider the following example. Given two operations made by
13
           separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and
14
           an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consis-
15
           tency between these two operations, the file pointer may be updated by the amount
16
           requested (10 bytes) even if the amount accessed is zero bytes. (End of advice to
17
           users.)
18
19
      13.6.10
              Examples
20
21
      The examples in this section illustrate the application of the MPI consistency and semantics
22
      guarantees. These address
23
24
         • conflicting accesses on file handles obtained from a single collective open, and
25
         • all accesses on file handles obtained from two separate collective opens.
26
27
          The simplest way to achieve consistency for conflicting accesses is to obtain sequential
28
      consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10
29
      integers. If the latter, every element of b will be 5. If nonatomic mode is set, the results of
30
     the read are undefined.
^{31}
32
     /* Process 0 */
33
           i, a[10] ;
      int
34
           TRUE = 1;
      int
35
36
     for ( i=0;i<10;i++)</pre>
37
         a[i] = 5;
38
39
     MPI_File_open( MPI_COMM_WORLD, "workfile",
40
                       MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
41
     MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
42
     MPI_File_set_atomicity( fh0, TRUE ) ;
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status) ;
43
44
     /* MPI_Barrier( MPI_COMM_WORLD ) ; */
45
      /* Process 1 */
46
47
      int
           b[10] ;
      int TRUE = 1;
48
```

MPI\_File\_open( MPI\_COMM\_WORLD, "workfile", MPI\_MODE\_RDWR | MPI\_MODE\_CREATE, MPI\_INFO\_NULL, &fh1 ); MPI\_File\_set\_view( fh1, 0, MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL ) ; MPI\_File\_set\_atomicity( fh1, TRUE ) ; /\* MPI\_Barrier( MPI\_COMM\_WORLD ) ; \*/ MPI\_File\_read\_at(fh1, 0, b, 10, MPI\_INT, &status) ;

A user may guarantee that the write on process 0 precedes the read on process 1 by imposing temporal order with, for example, calls to MPI\_BARRIER.

Advice to users. Routines other than MPI\_BARRIER may be used to impose temporal order. In the example above, process 0 could use MPI\_SEND to send a 0 byte message, received by process 1 using MPI\_RECV. (End of advice to users.)

Alternatively, a user can impose consistency with nonatomic mode set:

```
/* Process 0 */
int i, a[10] ;
for ( i=0;i<10;i++)</pre>
   a[i] = 5;
MPI_File_open( MPI_COMM_WORLD, "workfile",
                                                                                  21
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
                                                                                  22
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                  23
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh0 ) ;
                                                                                  27
                                                                                  28
/* Process 1 */
                                                                                  29
int b[10];
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_sync( fh1 ) ;
                                                                                  34
MPI_Barrier( MPI_COMM_WORLD ) ;
                                                                                  35
MPI_File_sync( fh1 ) ;
                                                                                  36
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
                                                                                  37
```

The "sync-barrier-sync" construct is required because:

- The barrier ensures that the write on process 0 occurs before the read on process 1.
- The first sync guarantees that the data written by all processes is transferred to the storage device.
- The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)

The following program represents an erroneous attempt to achieve consistency by eliminating the apparently superfluous second "sync" call for each process.

# **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9 10

11

12

13 14

15

16

1718

19 20

2425

26

30

31

32

33

38

39 40

41

42

43 44

45

46 47

```
1
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
\mathbf{2}
     /* Process 0 */
3
     int i, a[10] ;
4
     for ( i=0;i<10;i++)</pre>
\mathbf{5}
        a[i] = 5;
6
7
     MPI_File_open( MPI_COMM_WORLD, "workfile",
8
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
9
     MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
10
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
11
     MPI_File_sync( fh0 ) ;
12
     MPI_Barrier( MPI_COMM_WORLD ) ;
13
     /* Process 1 */
14
     int b[10];
15
16
     MPI_File_open( MPI_COMM_WORLD, "workfile",
17
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
     MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
18
19
     MPI_Barrier( MPI_COMM_WORLD ) ;
     MPI_File_sync( fh1 ) ;
20
21
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
22
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
23
24
     The above program also violates the MPI rule against out-of-order collective operations and
25
     will deadlock for implementations in which MPI_FILE_SYNC blocks.
26
27
          Advice to users. Some implementations may choose to implement MPI_FILE_SYNC
28
          as a temporally synchronizing function. When using such an implementation, the
29
           "sync-barrier-sync" construct above can be replaced by a single "sync." The results of
30
          using such code with an implementation for which MPI_FILE_SYNC is not temporally
^{31}
          synchronizing is undefined. (End of advice to users.)
32
33
     Asynchronous I/O
34
35
     The behavior of asynchronous I/O operations is determined by applying the rules specified
36
     above for synchronous I/O operations.
37
         The following examples all access a preexisting file "myfile." Word 10 in myfile initially
38
     contains the integer 2. Each example writes and reads word 10.
39
         First consider the following code fragment:
40
41
     int a = 4, b, TRUE=1;
42
     MPI_File_open( MPI_COMM_WORLD, "myfile",
43
                     MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
     MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
44
45
     /* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
46
     MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
47
     MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
48
     MPI_Waitall(2, reqs, statuses) ;
```

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
9
int a = 4, b;
                                                                                       10
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                       11
                MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                       12
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                       13
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
                                                                                       14
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
                                                                                       15
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
                                                                                       16
MPI_Wait(&regs[0], &status) ;
                                                                                       17
MPI_Wait(&reqs[1], &status) ;
                                                                                       18
                                                                                       19
If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee
                                                                                       20
sequential consistency in nonatomic mode.
                                                                                       21
    On the other hand, the following code fragment:
                                                                                       22
int a = 4, b;
                                                                                       23
                                                                                       ^{24}
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                       25
                MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                       26
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
                                                                                       27
MPI_Wait(&reqs[0], &status) ;
                                                                                       28
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
                                                                                       29
                                                                                       30
MPI_Wait(&reqs[1], &status) ;
                                                                                       31
defines the same ordering as:
                                                                                       32
                                                                                       33
int a = 4, b;
                                                                                       34
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                       35
                MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                       36
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
                                                                                       37
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status ) ;
                                                                                       38
MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status ) ;
                                                                                       39
Since
                                                                                       40
                                                                                       41
   • nonconcurrent operations on a single file handle are sequentially consistent, and
                                                                                       42
                                                                                       43
   • the program fragments specify an order for the operations,
                                                                                       44
MPI guarantees that both program fragments will read the value 4 into b. There is no need
                                                                                       45
to set atomic mode for this example.
                                                                                       46
    Similar considerations apply to conflicting accesses of the form:
                                                                                       47
                                                                                       48
```

1 2

3

4

5 6

7

```
<sup>1</sup> MPI_File_write_all_begin(fh,...);
<sup>2</sup> MPI_File_iread(fh,...);
<sup>3</sup> MPI_Wait(fh,...);
```

```
<sup>4</sup> MPI_File_write_all_end(fh,...) ;
```

Recall that constraints governing consistency and semantics are not relevant to the following:

```
8 MPI_File_write_all_begin(fh,...);
```

```
9 MPI_File_read_all_begin(fh,...) ;
```

MPI\_File\_read\_all\_end(fh,...);

MPI\_File\_write\_all\_end(fh,...) ;

<sup>12</sup> since split collective operations on the same file handle may not overlap (see Section 13.4.5, <sup>13</sup> page 526).

 $14 \\ 15$ 

16

22

23

 $^{24}$ 

25 26

27

43

5

6

7

```
13.7 I/O Error Handling
```

<sup>17</sup>By default, communication errors are fatal—MPI\_ERRORS\_ARE\_FATAL is the default error handler associated with MPI\_COMM\_WORLD. I/O errors are usually less catastrophic (e.g., "file not found") than communication errors, and common practice is to catch these errors and continue executing. For this reason, MPI provides additional error facilities for I/O.

Advice to users. MPI does not specify the state of a computation after an erroneous MPI call has occurred. A high-quality implementation will support the I/O error handling facilities, allowing users to write programs using common practice for I/O. (End of advice to users.)

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 8.3, page 342.

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,

<sup>33</sup> I/O error handling differs from communication error handling in another important <sup>34</sup> aspect. By default, the predefined error handler for file handles is MPI\_ERRORS\_RETURN. <sup>35</sup> The default file error handler has two purposes: when a new file handle is created (by <sup>36</sup> MPI\_FILE\_OPEN), the error handler for the new file handle is initially set to the default <sup>37</sup> error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., <sup>38</sup> MPI\_FILE\_OPEN or MPI\_FILE\_DELETE) use the default file error handler. The default <sup>39</sup> file error handler can be changed by specifying MPI\_FILE\_NULL as the

<sup>40</sup> fh argument to MPI\_FILE\_SET\_ERRHANDLER. The current value of the default file error <sup>41</sup> handler can be determined by passing MPI\_FILE\_NULL as the fh argument to <sup>42</sup> MPI\_FILE\_GET\_ERRHANDLER.

*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.*)

# 13.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI\_ERR\_TYPE.

MPI_ERR_FILE	Invalid file handle	7
MPI_ERR_NOT_SAME	Collective argument not identical on all	8
	processes, or collective routines called in	9
	a different order by different processes	10
MPI_ERR_AMODE	Error related to the amode passed to	11
	MPI_FILE_OPEN	12
MPI_ERR_UNSUPPORTED_DATARE	• Unsupported datarep passed to	13
	MPI_FILE_SET_VIEW	14
MPI_ERR_UNSUPPORTED_OPERAT	ON Unsupported operation, such as seeking on	15 16
	a file which supports sequential access only	
MPI_ERR_NO_SUCH_FILE	File does not exist	17
MPI_ERR_FILE_EXISTS	File exists	18
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	19
MPI_ERR_ACCESS	Permission denied	20
MPI_ERR_NO_SPACE	Not enough space	21
MPI_ERR_QUOTA	Quota exceeded	22
MPI_ERR_READ_ONLY	Read-only file or file system	23
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	24
	the file is currently open by some process	25
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	26
	tered because a data representation identi-	27
	fier that was already defined was passed to	28
	MPI_REGISTER_DATAREP	29
MPI_ERR_CONVERSION	An error occurred in a user supplied data	30
	conversion function.	31
MPI_ERR_IO	Other I/O error	32
		33
Table	13.3: I/O Error Classes	34
		35
		36
13.9 Examples		37
		38
13.9.1 Double Buffering with Split Collective I/O		39 40
This example shows how to even long	emputation and output. The computation is performed	41
This example shows how to overlap computation and output. The computation is performed		42
by the function compute_buffer().		43
/*=====================================		44
*		45
	_buffer	46
	_~~-~~	

\* Synopsis:

1 2

3

4

 $\mathbf{5}$ 

6

47

```
1
     *
          void double_buffer(
\mathbf{2}
     *
                    MPI_File fh,
                                                            ** IN
3
     *
                    MPI_Datatype buftype,
                                                            ** IN
4
     *
                    int bufcount
                                                            ** IN
5
     *
            )
6
     *
7
     * Description:
            Performs the steps to overlap computation with a collective write
8
     *
9
            by using a double-buffering technique.
     *
10
     *
11
     * Parameters:
12
     *
            fh
                               previously opened MPI file handle
                            MPI datatype for memory layout
13
            buftype
     *
14
     *
                              (Assumes a compatible view has been set on fh)
15
                             # buftype elements to transfer
     *
            bufcount
16
     *-----*/
17
18
     /* this macro switches which buffer "x" is pointing to */
19
    #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
20
21
    void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount)
22
     ſ
23
^{24}
       MPI_Status status; /* status for MPI calls */
       float *buffer1, *buffer2; /* buffers to hold results */
25
26
       float *compute_buf_ptr; /* destination buffer */
27
                                  /* for computing */
       float *write_buf_ptr; /* source for writing */
28
29
                                 /* determines when to quit */
       int done;
30
^{31}
       /* buffer initialization */
32
       buffer1 = (float *)
33
                         malloc(bufcount*sizeof(float)) ;
34
       buffer2 = (float *)
35
                          malloc(bufcount*sizeof(float)) ;
36
       compute_buf_ptr = buffer1 ; /* initially point to buffer1 */
37
       write_buf_ptr = buffer1 ; /* initially point to buffer1 */
38
39
40
       /* DOUBLE-BUFFER prolog:
41
            compute buffer1; then initiate writing buffer1 to disk
        *
42
        */
43
       compute_buffer(compute_buf_ptr, bufcount, &done);
44
       MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
45
       /* DOUBLE-BUFFER steady state:
46
47
        * Overlap writing old results from buffer pointed to by write_buf_ptr
48
        * with computing new results into buffer pointed to by compute_buf_ptr.
```

```
*
    *
       There is always one write-buffer and one compute-buffer in use
       during steady state.
    *
    */
  while (!done) {
      TOGGLE_PTR(compute_buf_ptr);
      compute_buffer(compute_buf_ptr, bufcount, &done);
      MPI_File_write_all_end(fh, write_buf_ptr, &status);
      TOGGLE_PTR(write_buf_ptr);
      MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
  }
   /* DOUBLE-BUFFER epilog:
    *
        wait for final write to complete.
    */
  MPI_File_write_all_end(fh, write_buf_ptr, &status);
  /* buffer cleanup */
  free(buffer1);
  free(buffer2);
}
```



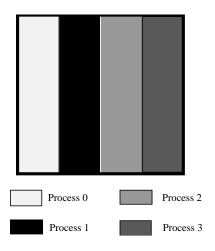


Figure 13.4: Example array file layout

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 Section 4.1.3 on page 95):

```
double subarray[100][25];
```

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $46 \\ 47$ 

```
1
\mathbf{2}
3
4
5
6
7
8
9
                                         MPI_DOUBLE
                                                           Holes
10
11
                       Figure 13.5: Example local array filetype for process 1
12
13
         MPI_Datatype filetype;
14
         int sizes[2], subsizes[2], starts[2];
15
         int rank;
16
17
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18
         sizes[0]=100; sizes[1]=100;
19
         subsizes[0]=100; subsizes[1]=25;
20
         starts[0]=0; starts[1]=rank*subsizes[1];
21
22
         MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
23
                                     MPI_DOUBLE, &filetype);
^{24}
25
          Or, equivalently in Fortran:
26
27
             double precision subarray(100,25)
28
             integer filetype, rank, ierror
29
             integer sizes(2), subsizes(2), starts(2)
30
^{31}
             call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
32
             sizes(1)=100
33
             sizes(2)=100
34
             subsizes(1)=100
35
             subsizes(2)=25
36
             starts(1)=0
37
             starts(2)=rank*subsizes(2)
38
39
             call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
40
                          MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                                &
41
                          filetype, ierror)
42
43
          The generated filetype will then describe the portion of the file contained within the
44
     process's subarray with holes for the space taken by the other processes. Figure 13.5 shows
45
     the filetype created for process 1.
46
47
48
```

# 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 607.

- 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.

 $^{31}$ 

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.

- 5. provide a no-op routine MPI\_PCONTROL in the MPI library.
- 12 13 14

15

1

2

3

4

5

6

7

8

9

10

11

# 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.

<sup>19</sup>Since MPI is a machine independent standard with many different implementations, <sup>20</sup>it is unreasonable to expect that the authors of profiling tools for MPI will have access to <sup>21</sup>the source code that implements MPI on any particular machine. It is therefore necessary <sup>22</sup>to provide a mechanism by which the implementors of such tools can collect whatever <sup>23</sup>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

# <sup>16</sup> 14.2.3 Logic of the Design

<sup>47</sup> Provided that an MPI implementation meets the requirements above, it is possible for
 <sup>48</sup> the implementor of the profiling system to intercept the MPI calls that are made by the

# Unofficial Draft for Comment Only

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	g level (integer)
int	MPI_Pcontrol(co	nst int l	evel,)	

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

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

# 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;
8
     static double totalTime = 0.0;
9
10
     int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
11
                   int dest, int tag, MPI_Comm comm)
12
     {
        13
14
        int extent;
15
                       = PMPI_Send(buffer,count,datatype,dest,tag,comm);
        int result
16
17
        MPI_Type_size(datatype, &extent); /* Compute size */
18
        totalBytes += count*extent;
19
20
        totalTime += MPI_Wtime() - tstart;
                                                        /* and time
                                                                               */
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 definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library), however if no other definition exists, then the linker will use the weak definition.

# Unofficial Draft for Comment Only

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 pre-processor 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're called from collective functions?"), we have decided not to enforce any restrictions on the author of the MPI library that would

# Unofficial Draft for Comment Only

 $\frac{4}{5}$ 

 $^{24}$ 

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 **ar**ed out of the base library and into the profiling one.

27

# 28 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 607.

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.

<sup>44</sup> Since one of the objectives of MPI is to permit efficient, low latency implementations, and <sup>45</sup> it is not the business of a standard to require a particular implementation language, we <sup>46</sup> 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

# Unofficial Draft for Comment Only

before calling the underlying MPI function. This capability has been demonstrated in the  $P^N$ MPI tool infrastructure [52].

# 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

### Unofficial Draft for Comment Only

1 2

3 4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

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 perfor-7 mance information through a set of control and performance variables defined by the MPI 8 implementation. Since some implementations may export a large number of variables, 9 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
	د۲	

Table 14.1: MPI tool information interface verbosity levels.

### 28 29

# 30

#### Binding MPI Tool Information Interface Variables to MPI Objects 14.3.2

 $^{31}$ 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

# **Unofficial Draft for Comment Only**

1

2

3

4 5

6

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. 

# 14.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization routines.

MPI_T_INIT_THREAD	(required,	provided)	)
-------------------	------------	-----------	---

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)

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-5izes 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

35 36

37

45

21

22

23

24

25 26

27

28

29

30

34

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

MPI\_T\_INIT\_THREAD up to the current point of execution, the MPI tool information in terface is no longer initialized. The interface can be reinitialized by subsequent calls to
 MPI\_T\_INIT\_THREAD.

At the end of the program execution, unless MPI\_ABORT is called, an application must have called MPI\_T\_INIT\_THREAD and MPI\_T\_FINALIZE an equal number of times.

### 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

### Unofficial Draft for Comment Only

 $\overline{7}$ 

-	sents a particular state. Each each that can be queried using MP	enumeration type can have $N$ different values, with I_T_ENUM_GET_INFO.
MPI_T_EN	IUM_GET_INFO(enumtype, nu	ım, 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_T	_enum_get_info(MPI_T_enum *name_len)	n enumtype, int *num, char *name, int
by this enu must repre The a as describe The re unique wit Name using MPL	meration typeas well as its nar sent at least one value. rguments name and name_len ed in Section 14.3.3. putine is required to return a h respect to all other names for s associated with individual va _T_ENUM_GET_ITEM.	his routine returns the number of items represented ne. $N$ must be greater than 0, i.e., the enumeration are used to return the name of the enumerations name of at least length one. This name must be or enumerations that the MPI implementation uses. alues in each enumeration enumtype can be queried
IN	IUM_GET_ITEM(enumtype, in enumtype	enumeration to be queried (handle)
IN	index	number of the value to be queried in this enumeration (integer)
OUT	value	variable value (integer)
OUT	name	buffer to return the string containing the name of the enumeration item (string)
INOUT	name_len	length of the string and/or buffer for $name\xspace$ (integer)
int MPI_T	_enum_get_item(MPI_T_enum *name, int *name_len	n enumtype, int index, int *value, char )
item as des If com enumeration	scribed in Section 14.3.3. apleted successfully, the routin on at the specified index. The . This name must be unique w	are used to return the name of the enumeration he returns the name/value pair that describes the call is further required to return a name of at least ith respect to all other names of items for the same

 $^{23}$ 

### 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 variables become 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

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 2	MPI_T_C\	/AR_GET_INFO(cvar_index, i desc_len, bind, scope)	name, 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)
10 11 12	OUT	datatype	MPI datatype of the information stored in the control variable (handle)
13 14	OUT	enumtype	optional descriptor for enumeration information (han- dle)
15 16 17	OUT	desc	buffer to return the string containing a description of the control variable (string)
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_7	<pre>*verbosity, MPI_Dat</pre>	_index, char *name, int *name_len, int atype *datatype, MPI_T_enum *enumtype, char n, int *bind, int *scope)
28 29 30 31 32 33	calls to the informatio The a as describe	is routine that query informa n. An MPI implementation i rguments name and name_lea ed in Section 14.3.3.	VAR_GET_INFO for a particular variable, subsequent ation about the same variable must return the same is not allowed to alter any of the returned values. In are used to return the name of the control variable time is required to return a name of at least length
34			respect to all other names for control variables used
35	by the MPI implementation.		
36 37	The argument verboard returns the verboard return of the variable (see beetion 14.9.1).		
38	The argument datatype returns the MPI datatype that is used to represent the control variable.		
39	If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the		
40	-	-	return it in enumtype. In this case, MPI returns an
41 42	enumeration identifier, which can then be used to gather more information as described in		
43	Section 14.3.5. If the datatype is not MPI_INT or the argument enumtype is the constant MPI_T_ENUM_NULL_no enumeration type is returned		
44	MPI_T_ENUM_NULL, no enumeration type is returned. The arguments desc and desc_len are used to return a description of the control variable		
45	as described in Section 14.3.3.		
46			al. If an MPI implementation does not to return a
47 48	description, the first character for desc must be set to the null character and desc_len must		
48	be set to c	one at the return of this call.	

# Unofficial Draft for Comment Only

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;
```

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{31}$ 

```
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);
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, MPI_T_ENUM_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 variables 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_C	VAR_READ(handle, buf)	
IN	handle	handle to the control variable to be read (handle)
OUT	buf	initial address of storage location for variable value
		(choice)

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 7 MPI\_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) 1213 int 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 18 returned 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 28variable 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 30 returns 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; 4445/\* This example assumes that the variable index \*/ 46/\* can be bound to a communicator \*/ 47 48

```
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;
```

14.3.7 Performance Variables

}

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 reseting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (*End of advice to implementors.*)

The classes are defined by the following constants:

• MPI\_T\_PVAR\_CLASS\_STATE

A performance variable in this class represents a set of discrete states. Variables of <sup>43</sup> this class are represented by MPI\_INT and can be set by the MPI implementation at <sup>44</sup> any time. Variables of this type should be described further using an enumeration, as <sup>45</sup> 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 <sup>47</sup> that variables of this class cannot overflow. <sup>48</sup>

# Unofficial Draft for Comment Only

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

# • MPI\_T\_PVAR\_CLASS\_LEVEL A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

# • MPI\_T\_PVAR\_CLASS\_SIZE

A performance variable in this class represents a value that is the fixed size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

18 19

# • MPI\_T\_PVAR\_CLASS\_PERCENTAGE

The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an MPI\_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

27

# • MPI\_T\_PVAR\_CLASS\_HIGHWATERMARK

A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is non-negative and grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_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\_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 nonnegative 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

A performance variable in this class counts the number of occurrences of a specific event (e.g., the number of memory allocations within an MPI library). The value of

# Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

20

21

22

23

24

25

26

28

29

30

31

32

33

34

35 36

37

38

39

40

41

42

43

44

 $45 \\ 46$ 

47

a variable of this class increases monotonically from the initialization or reset of the performance variable by one for each specific event that is observed. Values must 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\_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, 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 variables become inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, N:

### **Unofficial Draft for Comment Only**

 $\overline{7}$ 

 $^{24}$ 

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 function $MPI\_T\_PVAR\_GET\_INFO$ provides access to additional information for				
7 8	each varia	ble.				
9						
10 11	MPI_I_P\	MPI_T_PVAR_GET_INFO(pvar_index, name, name_len, verbosity, varclass, datatype, enum- type, 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 $name\xspace$ (integer)			
18	OUT	verbosity	verbosity level of this variable (integer)			
19 20	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 $desc\xspace$ (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 41 42	int MPI_1	<pre>int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len, int</pre>				
43 44 45 46 47 48	calls to the informatio The a variable as	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.				

<sup>48</sup> to return a name of at least length one.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1). The class of the performance variable is returned in the parameter var\_class. The class must be one of the constants defined in Section 14.3.7.

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 as described in Section 14.3.5 to gather more information. If the datatype is not MPI\_INT or the argument enumtype is the constant MPI\_T\_ENUM\_NULL, no emumeration type is returned.

Returning a description is optional. If an MPI implementation 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 **readon**ly 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.

### **Unofficial Draft for Comment Only**

1	MPI_T_PVAR_SESSION_FREE(session)				
2 3	INOUT	session	identifier of performance experiment session (handle)		
4 5	int MPI_T	_pvar_session_free(MPI_T_	pvar_session *session)		
6 7 8 9	longer be n	9	Calls to the MPI tool information interface can no ession after it is freed. On a successful return, MPI R_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	<pre>int MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,</pre>				
30 31 32 33 34 35 36 37 38	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.				
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	MPI_0 librar addre	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_PVAR_HANDLE_ALLOC. ( <i>End of advice to users.</i> )			

The value of index should be in the range 0 to $num_pvar - 1$ , where <sup>1</sup>					
num_pvar is the number of available performance variables as determined from a prior call <sup>2</sup>					
	to MPI_T_PVAR_GET_NUM. The type of the MPI object it references must be consistent <sup>3</sup> with the type returned in the bind argument in a prior call to MPI_T_PVAR_GET_INEO <sup>4</sup>				
ě		nent in a prior call to MPI_T_PVAR_GET_INFO. ls MPI_T_BIND_NO_OBJECT, the argument	5		
obj_handle		as wri_i_bind_no_object, the argument	6		
obj_nanale	is ignored.		7		
			8		
MPI_T_PV	'AR_HANDLE_FREE(session, h	handle)	9		
IN	session	identifier of performance experiment session (handle)	10		
INOUT	handle	handle to be freed (handle)	11 12		
int MPI_T	_pvar_handle_free(MPI_T_p *handle)	ovar_session session, MPI_T_pvar_handle	13 14 15		
When	a handle is no longer needed.	a user of the MPI tool information interface should	16		
	8	ee the handle in the session identified by the pa-	17		
rameter se	ssion and the associated resou	rces in the MPI implementation. On a successful	18		
$\operatorname{return},MF$	$PI$ sets the handle to $MPI\_T\_PV$	/AR_HANDLE_NULL.	19		
			20		
Starting an	d Stopping of Performance Var	iables	21 22		
Performan	ce variables that have the co	ntinuous flag set during the query operation are	22		
		s been allocated. Such variables may be queried at	24		
any time,	but they cannot be started or	stopped by the user. All other variables are in a	25		
stopped sta	ate after their handle has been	allocated; their values are not updated until they	26		
have been	started by the user.		27		
			28		
MPI_T_PV	AR_START(session, handle)		29		
IN	session	identifier of performance experiment session (handle)	30 31		
		, , ,	32		
IN	handle	handle of a performance variable (handle)	33		
			34		
int MPI_I	_pvar_start(MP1_1_pvar_se	ession session, MPI_T_pvar_handle handle)	35		
This functions starts the performance variable with the handle identified by the pa-					
	ndle in the session identified b	· *	37		
		NDLES is passed in handle, the MPI implementation	38		
attempts to start all variables within the session identified by the parameter session for <sup>39</sup>					
which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all 40					
variables are started successfully, otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned.					
	Continuous variables and variables that are already started are ignored when 4 MPI_T_PVAR_ALL_HANDLES is specified. 4				
IVII I_I_F V <i>F</i>	in_mer_inmiders is specified.		43 44		
			45		
			46		
			47		
48					

	MPI_T_PV	AR_STOP(session, handle)		
2 3	IN	session	identifier of performance experiment session (handle)	
4	IN	handle	handle of a performance variable (handle)	
5 6 i	int MPI_T_	_pvar_stop(MPI_T_pvar_ses	sion session, MPI_T_pvar_handle handle)	
7 8 9 <del>6</del> 10 11 8 12 V 13 V 13 V 14 N 15 N	This functions stops the performance variable with the handle identified by the param- eter handle in the session identified by the parameter session. If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to stop 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 stopped successfully, otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when MPI_T_PVAR_ALL_HANDLES is specified. Performance Variable Access Functions			
19				
20	MPI_T_PV	AR_READ(session, handle, buf	)	
21	IN	session	identifier of performance experiment session (handle)	
3	IN	handle	handle of a performance variable (handle)	
4 5	OUT	buf	initial address of storage location for variable value (choice)	
6 7 i 8	<pre>int MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>			
32 k 33 i 34 t 35 N	The MPI_T_PVAR_READ call queries the value of the performance variable with the handle handle in the session identified by the parameter session and stores the result in the buffer identified by the parameter buf. The user is responsible to 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). The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_READ.			
20	MPLT PV	AR_WRITE(session,handle, buf	.)	
10	IN	session	identifier of performance experiment session (handle)	
1 2	IN	handle	handle of a performance variable (handle)	
3 4	IN	buf	initial address of storage location for variable value (choice)	
15 16 17 18	int MPI_T_	_pvar_write(MPI_T_pvar_se const void* buf)	ssion session, MPI_T_pvar_handle handle,	

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. Read-only 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	32
		(choice)	34

### 

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
45
46
47
48

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

such 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.*)

16 E

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

```
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
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
index=-1;
                                                                           13
                                                                           14
i=0:
                                                                           15
while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
        /* Pass a buffer that is at least one character longer than */ ^{16}
                                                                           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. */
                                                                           20
        namelen=18;
                                                                           21
        err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
                 &var_class, &datatype, &enumtype, NULL, NULL, &bind,
                                                                           22
                                                                           23
                 &readonly, &continuous, &atomic);
                                                                           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);
                                                                           30
assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
                                                                           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
                                                                           36
if (err!=MPI_SUCCESS) return err;
                                                                           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;
```

Unofficial Draft for Comment Only

```
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
     resources such as memory allocations or other interactions with the operating system.
47
48
```

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 categorie	s (integer)

#### 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	ATEGORY_GET_INFO(cat_ind num_categories)	lex, name, name_len, desc, desc_len, num_cvars, num_pvars,	
$\frac{3}{4}$	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 $desc$ (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)	
15 16 17	OUT	num_categories	number of categories contained in the category (integer)	
19 20 21 22		<pre>*desc, int *desc_len *num_categories)</pre>	at_index, char *name, int *name_len, char a, int *num_cvars, int *num_pvars, int	
23 24 25 26 27	The arguments name and name_len are used to return the name of the category as described in Section 14.3.3. The routine is required to return a name of at least length one. This name must be unique with respect to all other names for categories used by the MPI implementation. The arguments desc and desc_len are used to return the description of the category as			
28 29 30 31 32	described in Section 14.3.3. Returning a description is optional. If an MPI implementation decides not to return a description, the first character for desc must be set to the null character and desc_len must be set to one at the return of this call. The function returns the number of control variables, performance variables and other			
32 33 34 35	-	contained in the queried cate ories, respectively.	egory in the arguments <code>num_cvars</code> , <code>num_pvars</code> , and	
36	MPI_T_CA	TEGORY_GET_CVARS(cat_i	ndex, 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	OUT	indices	an integer array of size len, indicating control variable indices (array of integers)	
44 45	int MPI_T	_category_get_cvars(int	<pre>cat_index, int len, int indices[])</pre>	
46 47 48			can be used to query which control variables are stegory contains zero or more control variables.	

MPI_T_CATEGORY_GET_PVARS(cat_index,len,indices) <sup>1</sup>					
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	6		
		variable indices (array of integers)	7		
			8 9		
int MPI_T	_category_get_pvars(int c	<pre>at_index, int len, int indices[])</pre>	10		
		an be used to query which performance variables A category contains zero or more performance	11 12 13		
			14 15		
MPI_T_CA	TEGORY_GET_CATEGORIES	(cat_index,len,indices)	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	20 21		
		(array of integers)	21		
			23		
int MPI_T	_category_get_categories(	<pre>int cat_index, int len, int indices[])</pre>	24		
<ul> <li>MPI_T_CATEGORY_GET_CATEGORIES can be used to query which other categories</li> <li>are contained in a particular category. A category contains zero or more other categories. As mentioned above, MPI implementations can grow the number of categories as well</li> <li>as the number of variables or other categories within a category. In order to allow users</li> <li>of the MPI tool information interface to check quickly whether new categories have been</li> <li>added or new variables or categories have been added to a category, MPI maintains a</li> <li>virtual timestamp. This timestamp is monotonically increasing during the execution and is</li> <li>returned by the following function:</li> </ul>					
MPI_T_CA	TEGORY_CHANGED(stamp)		$\frac{34}{35}$		
OUT	stamp	a virtual time stamp to indicate the last change to the categories (integer)	36 37 38		
int MDT T	_category_changed(int *st	2002)	39		
		-	40		
	If two subsequent calls to this routine return the same timestamp, it is guaranteed that the category information has not changed between the two calls. If the timestamp retrieved				
0		e categories have been added or expanded.	43		
		Gran and a second se	44		
	Advice to users. The timestamp value is purely virtual and only intended to check $4^{\pm}$				
		tion. It should not be used for any other purpose.	46		
(End of advice to users.)			47 48		
0±					

### Unofficial Draft for Comment Only

The index values returned in indices by MPI\_T\_CATEGORY\_GET\_CVARS, MPI\_T\_CATEGORY\_GET\_PVARS and MPI\_T\_CATEGORY\_GET\_CATEGORIES can be used as input to MPI\_T\_CVAR\_GET\_INFO, MPI\_T\_PVAR\_GET\_INFO and MPI\_T\_CATEGORY\_GET\_INFO, respectively. The user is responsible for allocating the arrays passed into the functions MPI\_T\_CATEGORY\_GET\_CVARS, MPI\_T\_CATEGORY\_GET\_PVARS and MPI\_T\_CATEGORY\_GET\_CATEGORIES. Starting from array index 0, each function writes up to len elements into the array. If the category contains more than len elements, the function returns an arbitrary subset of size len. Otherwise, the entire set of elements is returned in the beginning entries of the array, and any remaining array entries are not modified. 14.3.9Return Codes for the MPI tool information interface All functions defined as part of the MPI tool information interface return an integer error code (see Table 14.5) to indicate whether the function was completed successfully or was aborted. In the latter case the error code indicates the reason for not completing the routine. Such errors neither impact the execution of the MPI process nor invoke MPI error handlers. The MPI process continues executing regardless of the return code from the call. The MPI implementation is not required to check all user provided parameters; if a user passes invalid parameter values to any routine the behavior of the implementation is undefined. All error codes with the prefix MPI\_T\_ must be unique values and cannot overlap with any other error codes or error classes returned by the MPI implementation. Further, they shall be treated as MPI error classes as defined in Section 8.4 on page 349 and follow the same rules and restrictions. In particular, they must satisfy:  $0 = MPI_SUCCESS < MPI_T_ERR_... \le MPI_ERR_LASTCODE.$ Rationale. All MPI tool information interface functions must return error classes, because applications cannot portably call MPI\_ERROR\_CLASS before MPI\_INIT or MPI\_INIT\_THREAD to map an arbitrary error code to an error class. (End of rationale.) 14.3.10 Profiling Interface tion 14.2 apply equally to calls defined as part of the MPI tool information interface.

```
34
35
36
```

37

38

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 Sec-

- 39 40
- 41 42
- 43
- 44
- 45
- 46
- 47
- 48

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

1213

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

- 2526
- 2728

29

30

 $^{31}$ 32 33

1

Return Code	Description
Return Codes for All Functions in t	he 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 categ	ory 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 Varia	able Access and Control:
MPI_T_PVAR_{START STOP READ	) 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 Function	
MPI_T_ERR_INVALID_INDEX	The category index is invalid

### Chapter 15

## **Deprecated Functions**

### 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 655. The language bindings are modified.

 $^{31}$ 

#### MPI\_KEYVAL\_CREATE(copy\_fn, delete\_fn, keyval, extra\_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

# int MPI\_Keyval\_create(MPI\_Copy\_function \*copy\_fn, MPI\_Delete\_function \*delete\_fn, int \*keyval, void\* extra\_state)

For this routine, an interface within the mpi\_f08 module was never defined.

MPI_KEYVAL_CREATE(COPY_FN, DELE	TE_FN, KEYVAL	, EXTRA_STATE,	IERROR)	
EXTERNAL COPY_FN, DELETE_FN				
INTEGER KEYVAL, EXTRA_STATE	, IERROR			

The copy\_fn function is invoked when a communicator is duplicated by MPI\_COMM\_DUP. copy\_fn should be of type MPI\_Copy\_function, which is defined as follows:

	41
<pre>typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,</pre>	42
<pre>void *extra_state, void *attribute_val_in,</pre>	43
<pre>void *attribute_val_out, int *flag)</pre>	44
	45
A Fortran declaration for such a function is as follows:	46

For this routine, an interface within the mpi\_f08 module was never defined.

```
594
                                              CHAPTER 15. DEPRECATED FUNCTIONS
1
     SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
\mathbf{2}
                    ATTRIBUTE_VAL_OUT, FLAG, IERR)
3
          INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4
          ATTRIBUTE_VAL_OUT, IERR
5
          LOGICAL FLAG
6
          copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or
7
     FORTRAN; MPI_NULL_COPY_FN is a function that does nothing other than returning
8
     flag = 0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag =
9
     1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note
10
     that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.
11
          Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn
12
     function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call
13
     is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function,
14
     which is defined as follows:
15
16
     typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
17
     void *attribute_val, void *extra_state);
18
19
          A Fortran declaration for such a function is as follows:
20
     For this routine, an interface within the mpi_f08 module was never defined.
21
     SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
22
          INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
23
^{24}
          delete_fn may be specified as MPI_NULL_DELETE_FN from either C or FORTRAN;
25
     MPI_NULL_DELETE_FN is a function that does nothing, other than returning
26
     MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.
27
          The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL
28
     in MPI-2.0. The language independent definition of the deprecated function is the same as
29
     of the new function, except of the function name. The language bindings are modified.
30
^{31}
32
     MPI_KEYVAL_FREE(keyval)
33
       INOUT
                 keyval
                                             Frees the integer key value (integer)
34
35
     int MPI_Keyval_free(int *keyval)
36
37
     For this routine, an interface within the mpi_f08 module was never defined.
38
     MPI_KEYVAL_FREE(KEYVAL, IERROR)
39
          INTEGER KEYVAL, IERROR
40
41
          The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in
42
     MPI-2.0. The language independent definition of the deprecated function is the same as of
43
     the new function, except of the function name. The language bindings are modified.
44
45
46
47
48
```

MPI_ATTF	R_PUT(comm, keyval, attribute	e_val)	1	
INOUT	comm	communicator to which attribute will be attached (han- dle)	2 3 4	
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)	4 5 6	
IN	attribute_val	attribute value	7 8	
int MPI_A	ttr_put(MPI_Comm comm, i	nt keyval, void* attribute_val)	9 10	
For this ro	utine, an interface within the	<pre>mpi_f08 module was never defined.</pre>	11	
	PUT(COMM, KEYVAL, ATTRIB ER COMM, KEYVAL, ATTRIBU		12 13 14	
MPI-2.0. 1	The language independent def	d and is superseded by MPI_COMM_GET_ATTR in inition of the deprecated function is the same as of a name. The language bindings are modified.	14 15 16 17 18	
MPI_ATTE	R_GET(comm, keyval, attribute	e_val, flag)	19 20	
IN	comm	communicator to which attribute is attached (handle)	21	
IN	keyval	key value (integer)	22	
OUT	attribute_val	attribute value, unless $flag = false$	23 24	
OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key	24 25 26	
int MPT A	ttr get(MPI Comm comm, i	nt keyval, void *attribute_val, int *flag)	27 28	
	-	<pre>mpi_f08 module was never defined.</pre>	29	
		-	30 31	
	GET(COMM, KEYVAL, ATTRIB ER COMM, KEYVAL, ATTRIBU		32	
	CAL FLAG		33	
The fc	llowing function is deprecated	and is superseded by MPI_COMM_DELETE_ATTR	34	
		lefinition of the deprecated function is the same as	35 36	
	0 0 1	ion name. The language bindings are modified.	37	
			38	
MPI_ATTE	R_DELETE(comm, keyval)		39	
INOUT	comm	communicator to which attribute is attached (handle)	40 41	
IN	keyval	The key value of the deleted attribute (integer)	41	
	i cy vui	The key value of the deleted attribute (integer)	43	
int MPI_A	int MPI_Attr_delete(MPI_Comm comm, int keyval)			
		<pre>mpi_f08 module was never defined.</pre>	45	
			46 47	
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR) 48				

### Unofficial Draft for Comment Only

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 typedels 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_WIN_errhandler_fn MPI_FILe_errhandler_function MPI_WIN_errhandler_fn MPI_WIN_errhandler_function	1 2	IN	TEGER COMM, KEYVAL, IERROR	
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. $\frac{\text{Deprecated Name}}{\text{MPI_Comm_errhandler_fn}} \frac{\text{New Name}}{\text{MPI_Comm_errhandler_function}}$	3			
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	4	15.2	Deprecated since MPI-2.2	
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.         Image: Deprecated Name       New Name         Image: Deprecated Name       NPI_Comm_errhandler_function         Image: Deprecate Name       MPI_Win_errhandler_function         Image: Deprecate Name       MPI_Win_errhandler_function				
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_Win_errhandler_fn MPI_Win_errhandler_function MPI_Win_errhandler_fn MPI_Win_errhandler_function				s have been removed. See Chapter 16, Removed
<ul> <li>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.</li> <li>Deprecated Name New Name</li> <li>MPI_Comm_errhandler_fn MPI_Comm_errhandler_function</li> <li>MPI_Win_errhandler_fn MPI_Win_errhandler_function</li> </ul>				a been depresented and are supercaded by new
names were updated to match conventions of other function typedef names.           Deprecated Name         New Name           MPI_Comm_errhandler_fn         MPI_Comm_errhandler_function           MPI_Win_errhandler_fn         MPI_Win_errhandler_function           MPI_Win_errhandler_fn         MPI_Win_errhandler_function				
Deprecated Name         New Name           MPI_Comm_errhandler_fn         MPI_Comm_errhandler_function           MPI_Win_errhandler_fn         MPI_Win_errhandler_function           MPI_Win_errhandler_fn         MPI_Win_errhandler_function			• <b>•</b> · · · · · · · · · · · · · · · · · · ·	<b>a</b>
MPI_Comm_errhandler_fn         MPI_Comm_errhandler_function           MPI_Win_errhandler_fn         MPI_Win_errhandler_function           MPI_Win_errhandler_fn         MPI_Win_errhandler_fn           MPI_Win_errhandler_fn         MPI_Win_errhandler_fn           MPI_Win_errhandler_fn         MPI_Win_errhandler_fn           MPI_Win_errhandler_fn         MPI_Win_errhandler_fn           MPI_Win_errhandler_fn         MPI_Win_errhandler_fn		inclines	vere apaatoa to maten convention.	of other random typeder names.
MPI_Comm_errhandler_fn       MPI_Comm_errhandler_function         MPI_Win_errhandler_fn       MPI_Win_errhandler_function         MPI_Win_errhandler_fn       MPI_Win_errhandler_fin         MPI_Win_errhandler_fn       M	12		Deprecated Name	New Name
MP1_Win_errhandler_fn         MP1_Win_errhandler_function           III         IIII           IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII	13			MPI_Comm_errhandler_function
	14		MPI_File_errhandler_fn	MPI_File_errhandler_function
17         18         19         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         20         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         24         25         26         26	15		MPI_Win_errhandler_fn	MPI_Win_errhandler_function
18         19         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         24         24         24         24         25	16			
19         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         24         25         26         27         28	17			
20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         21         22         23	18			
21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24				
22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         29         20         21         22         23         24         25         26         27         28         29         29         29         20         21         22         23         24         24         25         26         27         28         29         20         21         22				
23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         20         21         22         23         24         25         26         27         28         29         21         22         23         24         25         26         27         28				
24         25         26         27         28         29         30         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         310         311         312         313         314         315         316         317         318				
26         27         28         29         30         31         32         33         34         35         36         37         38         39         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         310         311         312         313         314         314         315         316         317         318         319         310 <td></td> <td></td> <td></td> <td></td>				
28         29         30         31         32         33         34         35         36         37         38         39         31         32         33         34         35         36         37         38         39         31         32         33         34         35         36         37         38         39         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         310         32         33         34         35         36 <td>25</td> <td></td> <td></td> <td></td>	25			
28         29         30         31         32         33         34         35         36         37         38         39         39         31         32         33         34         35         36         37         38         39         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         31         32         33         34         35         36         37         38         39         310         32         33         34         35 <td>26</td> <td></td> <td></td> <td></td>	26			
29         30         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         40         41         42         43         44         45         46         47         48         49         41         42         43         44         45         46         47         48         49         41         42         43         44         45         46         47         48         49         41         42         43         44         45         46	27			
30         31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         40         41         42         43         44         45         46	28			
31         32         33         34         35         36         37         38         39         30         31         32         33         34         35         36         37         38         39         40         41         42         43         44         45         46	29			
32         33         34         35         36         37         38         39         40         41         42         43         44         45         46	30			
33         34         35         36         37         38         39         40         41         42         43         44         45         46	31			
34         35         36         37         38         39         40         41         42         43         44         45         46				
35         36         37         38         39         40         41         42         43         44         45         46				
36         37         38         39         40         41         42         43         44         45         46				
37         38         39         40         41         42         43         44         45         46				
38         39         40         41         42         43         44         45         46				
40         41         42         43         44         45         46				
41         42         43         44         45         46	39			
42 43 44 45 46	40			
43 44 45 46	41			
44 45 46	42			
45 46	43			
46				
4(				
48				

### 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.

MPI_ADDRESS MPI_GET_ADDRESS
MFI_ADDRESS MFI_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 ion page 598 shows the removed MPI-1 datatypes and their replacements.

	598CHAPTER 16. REMOVED INTERFACES
1 2 3 4	RemovedMPI-2 ReplacementMPI_LB (and MPI::LB)MPI_TYPE_CREATE_RESIZEDMPI_UB (and MPI::UB)MPI_TYPE_CREATE_RESIZED
5 6 7	Table 16.2: Removed MPI-1 datatypes and their replacements
8 9	16.1.4 Removed MPI-1 Constants
10	Table 16.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.
11	Removed MPI-1 Constants
12	C type: const int (or unnamed enum) C++ type:
13 14	Fortran type: INTEGER const int (or unnamed enum)
15	MPI_COMBINER_HINDEXED_INTEGER MPI::COMBINER_HINDEXED_INTEGER
16	MPI_COMBINER_HVECTOR_INTEGER MPI::COMBINER_HVECTOR_INTEGER
17	MPI_COMBINER_STRUCT_INTEGER MPI::COMBINER_STRUCT_INTEGER
18	
19	Table 16.3: Removed MPI-1 constants
20	
21	
22	16.1.5 Removed MPI-1 Callback Prototyes
23 24	Table 16.4 shows the removed MPI-1 callback prototypes and their MPI-2 replacements.
25	Removed MPI-2 Replacement
26	MPI_Handler_function MPI_Comm_errhandler_function
27	
28 29 30	Table 16.4: Removed MPI-1 callback prototypes and their replacements
31 32	16.2 C++ Bindings
33	The C++ bindings were deprecated as of MPI-2.2. The C++ bindings are removed in
34	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by
35	an implementation as described in the MPI-2.2 standard.
36 37	an implementation as described in the Wi 1-2.2 standard.
38	
39 40	
40 41	
41	
42	
43	
44	
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] + TR 29113 [42].

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 + TR 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 615. (End of rationale.)

MPI defines three methods of Fortran support:

- 1. USE mpi\_f08: This method is described in Section 17.1.2 and requires compiletime 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 + TR 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, and its use is therefore not recommended. It exists only for backwards compatibility.
- 3. **INCLUDE 'mpif.h':** This method is described in Section 17.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

1 Compliant MPI-3 implementations providing a Fortran interface must provide one or  $\mathbf{2}$ both of the following: 3 • The USE mpi\_f08 Fortran support method. 4 5• The USE mpi and INCLUDE 'mpif.h' Fortran support methods. 6  $\overline{7}$ Section 17.1.6 on page 611 describes restrictions if the compiler does not support all 8 the needed features. 9 Application subroutines and functions may use either one of the modules or the mpif.h 10 include file. An implementation may require the use of one of the modules to prevent type 11mismatch errors. 12Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h 13 enforces type checking on a particular system. Using a module provides several poten-14tial advantages over using an include file; the mpi\_f08 module offers the most robust 15and complete Fortran support. (End of advice to users.) 1617 In a single application, it must be possible to link together routines which USE mpi\_f08, 18 USE mpi, and INCLUDE mpif.h. 19The INTEGER compile-time constant MPI\_SUBARRAYS\_SUPPORTED is set to 20.TRUE. if all buffer choice arguments are defined in explicit interfaces with assumed-type 21and assumed-rank [42]; otherwise it is set to .FALSE.. The INTEGER compile-time constant 22MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 23added to the choice buffer arguments of all nonblocking interfaces **and** the underlying  $^{24}$ Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of 25TR 29113), otherwise it is set to .FALSE. These constants exist with each Fortran support 26method, but not in the C/C++ header files. The values may be different for each Fortran 27support method. All other constants and the integer values of handles must be the same 28for each Fortran support method. 29Section 17.1.2 through 17.1.4 define the Fortran support methods. The Fortran in-30 terfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding full  $^{31}$ interface specification together with the used linker names and implications for the pro-32 filing interface. Section 17.1.6 the implementation of the MPI routines for different ver-33 sions of the Fortran standard. Section 17.1.7 summarizes major requirements for valid 34MPI-3.0 implementations with Fortran support. Section 17.1.8 and Section 17.1.9 de-35 scribe additional functionality that is part of the Fortran support. MPI\_F\_SYNC\_REG 36 is needed for one of the methods to prevent register optimization problems. A set of func-37 tions provides additional support for Fortran intrinsic numeric types, including parameter-38 ized types: MPI\_SIZEOF, MPI\_TYPE\_MATCH\_SIZE, MPI\_TYPE\_CREATE\_F90\_INTEGER, 39 MPI\_TYPE\_CREATE\_F90\_REAL and MPI\_TYPE\_CREATE\_F90\_COMPLEX. In the context 40 of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 41 parameters. Sections 17.1.10 through 17.1.19 give an overview and details on known prob-42lems when using Fortran together with MPI; Section 17.1.20 compares the Fortran problems 43 with those in C. 44

45

### <sup>46</sup> 17.1.2 Fortran Support Through the mpi\_f08 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 611 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 603.

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 INTEGER compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TR 29113). See Section 17.1.6 on page 611 for older compiler versions.
- Set the INTEGER compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TR 29113 feature 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 TR 29113 feature is not needed for the support of noncontiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TR 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 17.1.6 on page 611 for details.

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

2829

30

31

32

33

34

35 36

37

38

39

41

47

48

• 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.)

If a dummy argument is declared with INTENT(OUT), then the Advice to users. 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 603. (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 Report "TR 29113 Further Interoperability with C" [42] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

40 Rationale. The features in TR 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 42higher level of integration between Fortran-specific features and C than was provided 43 in the Fortran 2008 standard; part of this design is based on requirements from the 44 MPI Forum to support MPI-3.0. According to [41] page iv, last paragraph, "it is 45the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified 46 by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve

### **Unofficial Draft for Comment Only**

proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations".

The TR 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).

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. TR 29113 also has a provision for OPTIONAL arguments in BIND(C) interfaces.

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 TR29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

### 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. Within all MPI function specifications, the second 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 and allows positional and keyword-based argument lists.
Define all MPI handles as type INTEGER.
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.

handles into new-style handles with a named type. (End of rationale.)

 $\mathbf{2}$ 

 $\overline{7}$ 

 $\frac{44}{45}$ 

• 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 INTEGER compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces and the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TR 29113), otherwise to .FALSE..

Advice to users. For an MPI implementation that fully supports nonblocking calls with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses 'contiguous' but not 'simply contiguous' ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constaints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Onother reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copyin/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 17.1.12 on page 628 for more details. (End of advice to users.)

- A high quality MPI implementation may enhance the interface by using TYPE(\*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !\$PRAGMA IGNORE\_TKR or a set of overloaded functions as described by M. Hennecke in [28], if the compiler supports this TR 29113 language feature. See Section 17.1.6 on page 611 for further details.
- Set the INTEGER compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(\*), DIMENSION(..), otherwise set it to .FALSE.. With MPI\_SUBARRAYS\_SUPPORTED==.TRUE., non-contiguous subarrays can be used as buffers in nonblocking routines.
- Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TR 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be disallowed. See Section 17.1.6 on page 611 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.

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 34

35

36

37 38

39

40

41

42

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi\_f08 bindings. (End of advice to implementors.)

Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran INTENT. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI\_BOTTOM with a dummy OUT argument. Moreover, "constants" such as MPI\_BOTTOM and MPI\_STATUS\_IGNORE are not constants as defined by Fortran, but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent 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 20argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-21TENT attribute, because INTENT(INOUT) always requires that the associated ac-22tual argument is *definable*". Applications that include mpif.h may not expect that 23INTENT (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.)

#### Fortran Support Through the mpif.h Include File 17.1.4

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

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:

39 • Define all named MPI constants. 40 • Declare MPI functions that return a value. 41 42• Define all handles as INTEGER. 43 44• Be valid and equivalent for both fixed and free source form. 45For each MPI routine, an implementation can choose to use an implicit or explicit interface 46for the second Fortran binding (in deprecated routines, the first one may be omitted). 47

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13 1415

16

17

18

19

 $^{24}$ 

25

26

27

282930

3132

33

34

35

36

37

38

1 2	• Set the INTEGER compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the mpi
3	module. In the case of implicit interfaces for choice buffer or nonblocking routines,
4	the constants must be set to .FALSE
5	
6	Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is
7	strongly encouraged for the following reasons:
8	• Most maif h implementations do not include compile time argument checking
9	• Most mpif.h implementations do not include compile-time argument checking.
10	• Therefore, too many bugs in MPI applications remain undetected at compile-
11	time, such as:
12	<ul> <li>Missing ierror as last argument in most Fortran bindings.</li> </ul>
13	- Declaration of a status as an INTEGER variable instead of an INTEGER array
14	with size MPI_STATUS_SIZE.
15	- Wrong argument positions; e.g., interchanging the count and
16	datatype arguments.
17	- Passing wrong MPI handles; e.g., passing a datatype instead of a communi-
18	cator.
19	• The migration from mpif.h to the mpi module should be relatively straightfor-
20	ward (i.e., substituting include 'mpif.h' after an implicit statement by use
21	mpi before such implicit statement) as long as the application syntax is correct.
22	• • • • • • • • • • •
23 24	• 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.
20	(End of advice to users.)
28	
29	Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to
30	retain strong backward compatibility. Internally, mpif.h and the mpi module may be
31	implemented so that the same (or similar) library implementation of the MPI routines
32	can be used. ( <i>End of rationale.</i> )
33	
34	Advice to implementors. To make mpif.h compatible with both fixed- and free-source
35	forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-
36	form line length, it is recommended that the requirement of usability in free and fixed
37	source form applications be met by constructing mpif.h without any continuation
38	lines. This should be possible because mpif.h may contain only declarations, and
39	because common block declarations can be split among several lines. The argument
40	names may need to be shortened to keep the SUBROUTINE statement within the allowed
41	72-6=66 characters, e.g.,
42	INTERFACE
43	SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
44	! dummy argument declarations
45	
46	This line has 65 characters and is the longest in MPI-3.0.
47 48	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
	and a prime of the second the second of the second second would not up

shorten their linker names in mpif.h, the mpif.h cannot set	
MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING $ m equals$	5
.TRUE., because such shortening is invalid. For example,	:
MPI_FILE_WRITE_AT_ALL_BEGIN with 6 arguments, may be defined:	
INTERFACE MPI_FILE_WRITE_AT_ALL_BEGIN	ţ

SUBROUTINE MPI\_X(a,b,c,d,e,f)BIND(C,NAME='MPI\_File\_write\_at\_all\_begin\_f')
... ! dummy argument declarations

This would need a line length of 73 characters, i.e., the C routine name 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 TR 29113. To support Fortran 77 as well as Fortran 90 and later, it may be necessary to eliminate all comments from mpif.h. (*End of advice to implementors.*)

### 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 given 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 TR 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(...)

• With the Fortran mpi module or mpif.h include file, if MPI\_SUBARRAYS\_SUPPORTED equals .FALSE.:

The linker name of each routine is defined through the linker-name mapping of the Fortran compiler. For example, MPI\_SEND may be mapped to mpi\_send\_. Example:

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 

$\frac{1}{2}$	INTERFACE SUBROUTINE MPI_SEND()
3 4 5	• With the Fortran mpi module or mpif.h include file, if MPI_SUBARRAYS_SUPPORTED equals .TRUE.:
6 7 8	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 _f suffix, e.g., MPI_Send_f. Prototype example:
9 10 11	<pre>INTERFACE SUBROUTINE MPI_SEND() BIND(C,NAME='MPI_Send_f')</pre>
12 13 14 15 16	If the support of subarrays is different for the mpi module and the mpif.h include file, then both linker-name methods can be used in the same application. If the application also uses the mpi_f08 module and was compiled with this module partially before and after the subarrays were supported, then all four interfaces are used within the same application.
17 18 19 20 21 22	Rationale. After a compiler provides the facilities from TR29113, i.e., TYPE(*), DIMENSION(), it is possible to change the bindings within a Fortran support method to support subarrays and without recompiling the complete application. Of course, only recompiled routines can benefit from the added facilities. There is no binary compatibility conflict because each interface uses its own linker names and all interfaces use the same constants and type definitions. (End of rationale.)
23 24 25 26 27 28 29 30	A user-written or middleware profiling routine that is written according to the same binding rules will have the same linker name, and therefore, can interpose itself as the MPI library routine. The profiling routine can internally call the matching PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments. In this case, the profiling software must use the same Fortran support method as used in the calling application program, because the C, mpi_f08 and mpi callback prototypes are different.
31 32 33 34 35	Advice to users. This advice is mainly for tool writers. Even if an MPI library supports subarrays in all three Fortran support methods, a portable profiling layer should also provide the two interfaces for MPI_SUBARRAYS_SUPPORTED==.FALSE. to support older binary user routines that were compiled before TR29113 level support was achieved.
36 37 38 39 40 41 42	If a user application calls MPI_SEND, then the chosen Fortran support method to- gether with the MPI implement decision about MPI_SUBARRAYS_SUPPORTED imply, to which linker name the compiler will translate this call, i.e., whether the applica- tion calls mpi_send, or MPI_Send_f, or mpi_send_f08, or MPI_Send_f08. If the profiling layer wants to be independent of the decision of the user program and MPI implementation, then it should provide all four routines. For example:
43 44 45 46	<pre>SUBROUTINE MPI_SEND() BIND(C,NAME='MPI_Send_f') USE mpi CALL PMPI_SEND() END SUBROUTINE</pre>
47 48	The MPI library must provide the PMPI_SEND routine according to the same rules as for providing the MPI_SEND routine. ( <i>End of advice to users.</i> )

Advice to implementors. If an implementation provides in a first step two sets of routines, one for the mpi module and mpif.h, and the other for the mpi\_f08 module, and both sets without TR 29113, i.e., MPI\_SUBARRAYS\_SUPPORTED equals .FALSE., and the implementor wants to add a TR 29113 based set of routines, then it is not necessary to add two full sets of routines. For full quality, it is enough to implement in each set only those routines that have a choice buffer argument. (End of advice to *implementors.*)

In the case that a Fortran binding consists of multiple routines through function overloading, the base names of overloaded routines are appended by a suffix notifying the difference in the argument list. For example, MPI\_ALLOC\_MEM (in the mpi module and mpif.h) has an INTEGER(KIND=...) baseptr argument without a suffix. This routine is overloaded by a routine with TYPE(C\_PTR) baseptr and the suffix \_CPTR. The implied linker name base is MPI\_ALLOC\_MEM\_CPTR. It is mapped to the linker names MPI\_Alloc\_mem\_cptr\_f, and, e.g., mpi\_alloc\_mem\_cptr\_\_. Note that these routines are always called via the interface name MPI ALLOC MEM by the application within all Fortran support methods.

For routines without ASYNCHRONOUS choice buffers and that are not predefined callback routines, the implementor can freely choose to implement the routines according to the rules for MPI\_SUBARRAYS\_SUPPORTED equals .TRUE. or .FALSE., provided that the following rule about routine grouping is fulfilled. The implementation of routines with ASYNCHRONOUS 20choice buffers depends on the rules for the provided Fortran support method and language 21level of the underlying compiler. Predefined callback routines for the mpi\_f08 module 22 must be implemented with BIND(C) interfaces, and for the mpi module and mpif.h without 23BIND(C).

Similar MPI routines are grouped together for linker symbol scheme classification. If the peer routine of a group is available within an MPI library with one of its possible linker names then all of the routines in this group must provided according to the same linker name scheme. If the peer routine is not available through a linker name scheme then all other routines in the group nust not be available through this scheme. Peer routines and their groups:

MPI_ALLOC_MEM	MPI_ALLOC_MEM and MPI_WIN_ALLOCATE.	31
MPI_FREE_MEM	Only this routine is in this group.	32
MPI_GET_ADDRESS	MPI_GET_ADDRESS and MPI_ADDRESS.	33
		34
MPI_SEND	All routines with choice buffer arguments that	35
	are not declared as ASYNCHRONOUS within the	36
	mpi_f08 module.	37
MPI_ISEND	All routines with choice buffer arguments	
	that are declared as ASYNCHRONOUS within the	38
	mpi_f08 module.	39
MPI_OP_CREATE	Only this routine is in this group.	40
MPI_REGISTER_DATAREP	Only this routine is in this group.	41
		42
MPI_COMM_KEYVAL_CREATE	All other routines with callback function argu-	43
	ments.	44
MPI_COMM_DUP_FN	All predefined callback routines.	45
MPI_COMM_RANK	All other MPI routines.	
		46

Additionally, four C preprocessor macros are available in mpi.h for each routine group. 47The name of the macros are the peer routine name written as in the list above and appended 48

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

24

25

26

27

28

29

1 with one of the following suffixes and meanings: 2 \_mpi\_f08\_BIND\_C The macro is set to 1 if the BIND(C) linker name with the 3 linker suffix \_f08 is available for all routines within this group 4 (e.g., MPI\_Send\_f08), otherwise it is set to 0. 5\_mpi\_f08\_BIND\_F The macro is set to 1 if the Fortran linker name with the 6 linker suffix \_f08 is available for all routines within this group 7 (e.g., mpi\_send\_f08\_\_), otherwise it is set to 0. 8 \_mpi\_BIND\_C ... The macro is set to 1 if the BIND(C) linker name with the 9 linker suffix \_f is available for all routines within this group 10 (e.g., MPI\_Send\_f), otherwise it is set to 0. 11 \_mpi\_BIND\_F ... The macro is set to 1 if the Fortran linker name without 12a linker suffix is available for all routines within this group 13 (e.g., mpi\_send\_\_), otherwise it is set to 0. 1415For example 16. . . 17 #define MPI\_SEND\_mpi\_f08\_BIND\_C 0 18 #define MPI\_SEND\_mpi\_f08\_BIND\_F 1 19 #define MPI\_SEND\_mpi\_BIND\_C 0 20#define MPI\_SEND\_mpi\_BIND\_F 1 2122 #define MPI\_ISEND\_mpi\_f08\_BIND\_C 1 23#define MPI\_ISEND\_mpi\_f08\_BIND\_F 1 24#define MPI\_ISEND\_mpi\_BIND\_C 1 25#define MPI\_ISEND\_mpi\_BIND\_F 1 26. . . 27#define MPI\_COMM\_DUP\_FN\_mpi\_f08\_BIND\_C 1 28 #define MPI\_COMM\_DUP\_FN\_mpi\_f08\_BIND\_F 0 29 #define MPI\_COMM\_DUP\_FN\_mpi\_BIND\_C 0 30 #define MPI\_COMM\_DUP\_FN\_mpi\_BIND\_F 1 31. . . 32 33 shows, that 34 • the routines in the MPI\_SEND group are only available through their Fortran linker 35 names (e.g., mpi\_send\_f08\_\_, mpi\_send\_\_, mpi\_recv\_f08\_\_, mpi\_recv\_\_, ...), 36 37 • the routines in the MPI\_ISEND group are available with all four interfaces: the MPI 38 library, the mpi\_f08 and mpi modules (that provide the TR 29113 quality), and this 39 MPI library supports application routines that are compiled with an older MPI library 40 version with \_BIND\_C set to 0 and \_BIND\_F set to 1. 41 42For the predefined callbacks, there is no choice, because the interfaces must fit to the 43 callback function prototypes which are BIND(C) based for mpi\_f08 and without BIND(C) 44for the mpi module and mpif.h. 4546Advice to implementors. If all following conditions are fulfilled (which is the case for 47 most compilers): 48

- the handles in the mpi\_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing 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 for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compiler,
- the compiler does not provide TR 29113,

then for most groups, the implementor may use the same internal routine implementations for all Fortran support methods but with several different linker names. For TR 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 TR 29113	Upgrade to TR 29113	Upgrade for strided data optimization	New impl. with TR 29113
MPI_ALLOC_MEM	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
MPI_FREE_MEM	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
MPI_GET_ADDRESS	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
MPI_SEND	0/1/0/1	0/1/0/1	1/1/1/1	1/0/1/0
MPI_ISEND	0/1/0/1	1/1/1/1	1/1/1/1	1/0/1/0
MPI_OP_CREATE	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
MPI_REGISTER_DATAREP	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
MPI_COMM_KEYVAL_CREATE	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
MPI_COMM_DUP_FN	1/0/0/1	1/0/0/1	1/0/0/1	1/0/0/1
MPI_COMM_RANK	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0

<sup>(</sup>End of advice to implementors.)

### 17.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
  - MPI identifiers are limited to thirty or more, not six, significant characters.
  - MPI identifiers may contain underscores after the first character.
  - An MPI subroutine with a choice argument may be called with different argument types.
  - Although not required b the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI\_ADDRESS and MPI\_BOTTOM may cause problems if an address

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10 11

12

13

14

15

16

31 32 33

34 35

36

37

38 39

40

 $41 \\ 42$ 

43

44

45

1 2	does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with $MPI_GET_ADDRESS$ , but not for Fortran 77.)
$\frac{3}{4}$	• For Fortran 90:
5	The major additional features that are needed from Fortran 90 are:
6	- The MODULE and INTERFACE concept.
7	- The KIND= and SELECTEDKIND concept.
8 9	<ul> <li>Fortran derived TYPEs and the SEQUENCE attribute.</li> </ul>
10	<ul> <li>The OPTIONAL attribute for dummy arguments.</li> </ul>
11	
12	<ul> <li>Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.</li> </ul>
13 14	
15	With these features, MPI-1.1 - MPI-2.2 can be implemented without restrictions. MPI- 3.0 can be implemented with some restrictions. The Fortran support methods are
16	abbreviated with $S1 = \text{the mpi_f08 module}$ , $S2 = \text{the mpi module}$ , and $S3 = \text{the}$
17	mpif.f include file. If not stated otherwise, restrictions exist for each method which
18	prevent implementing the complete semantics of MPI-3.0.
19 20	– MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-
21	contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,
22	or split-collective I/O.
23	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa-
24 25	tion is possible.
25 26	- In this preliminary interface of S1, the following changes are necessary:
27	* The routines are not BIND(C).
28	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
29	like !\$PRAGMA IGNORE_TKR.
30 31	* The ASYNCHRONOUS attribute is omitted.
32	* <b>PROCEDURE()</b> callback declarations are substituted by <b>EXTERNAL</b> .
33	- The linker names are specified in Section 17.1.5 on page 607.
34	- Due to the rules specified in Section 17.1.5 on page 607, choice buffer declarations
35	should be implemented only with non-standardized extensions like !\$PRAGMA
36 37	IGNORE_TKR (as long as $F2008+TR29113$ is not available).
38	In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different
39	MPI function for each possible buffer type (as mentioned in Section 17.1.11 on
40	page 627). Such overloading would also imply restrictions for passing Fortran
41	derived types as choice buffer, see also Section $17.1.15$ on page $631$ .
42 43	Only in S1: The implicit interfaces for routines with choice buffer arguments
44	imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommon dad not to provide the <b>mai</b> for module if such an extension is not
45	it is recommended not to provide the mpi_f08 module if such an extension is not available.
46	- The ASYNCHRONOUS attribute can <b>not</b> be used in applications to protect buffers
47	in nonblocking MPI calls (S1-S3).
48	

- The TYPE(C\_PTR) binding of the MPI\_ALLOC\_MEM and MPI\_WIN\_ALLOCATE routines is not available.
- In S1 and S2, the definition of the handle types (e.g., TYPE(MPI\_Comm) and the status type TYPE(MPI\_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi\_f08 module for Fortran 90 compilers. In this case, the mpi\_f08 handle types and all routines, constants and types ralated to TYPE(MPI\_Status) (see Section 17.2.5 on page 650) are also not available in the mpi module and mpif.h.

	13
For Fortran 95:	14
The quality of the MPI interface and the restrictions are the same as with Fortran 90.	15
	16
For Fortran 2003:	17
The major features that are needed from Fortran 2003 are:	18 19
– Interoperability with C, i.e.,	19 20
* BIND(C, NAME='') interfaces.	20
* BIND(C) derived types.	22
* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.	23
- The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy	24
arguments.	25
	26
- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O.	27
This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TR 29113.	28
Wit reominumeation in the 11t 25115.	29 30
With these features (but still without the features of TR29113), MPI-1.1 - MPI-2.2 $$	31
can be implemented without restrictions, but with one enhancement:	32
$-$ The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as	33
long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR	34
and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a	35
void * argument.	36
	37
MPI-3.0 can be implemented with the following restrictions:	38
— MPI_SUBARRAYS_SUPPORTED equals .FALSE	39
- For S1, only a preliminary implementation is possible. The following changes are	40 41
necessary:	41 42
* The routines are not BIND(C).	43
* TYPE(*), DIMENSION() is substituted by non-standardized extensions	44
like !\$PRAGMA IGNORE_TKR.	45
- The linker names are specified in Section 17.1.5 on page 607.	46
- The mixer names are specified in Section 17.1.5 on page 007.	47
	48

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

1	- With S1, the ASYNCHRONOUS is required as specified in the second Fortran inter-
2 3	faces. With $S2$ and $S3$ the implementation can also add this attribute if explicit interfaces are used.
4 5 6 7	<ul> <li>The ASYNCHRONOUS Fortran attribute can be used in applications to <i>try to</i> protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.</li> </ul>
8 9 10	<ul> <li>The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines can be used only for Fortran types that are C compatible.</li> </ul>
11 12	<ul> <li>The same restriction as for Fortran 90 applies if non-standardized extensions like !\$PRAGMA_IGNORE_TKR are not available.</li> </ul>
13 14 15	For Fortran $2008 + TR 29113$ and later and For Fortran $2003 + TR 29113$ :
16	The major feature that are needed from TR29113 are:
17	- TYPE(*), DIMENSION() is available.
18 19 20	- The <code>ASYNCHRONOUS</code> attribute is extended to protect also nonblocking <code>MPI</code> communication.
21	- OPTIONAL dummy arguments are allowed in combination with BIND(C) interfaces.
22 23	<ul> <li>CHARACTER(LEN=*) dummy arguments are allowed in combination with BIND(C) interfaces.</li> </ul>
24 25 26	<ul> <li>The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.</li> </ul>
27	Using these features, $MPI-3.0$ can be implemented without any restrictions.
28 29 30 31 32	<ul> <li>With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines can be used for any Fortran type.</li> </ul>
33 34 35 36	<ul> <li>With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation dependent. A high quality implementation will also provide MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.</li> </ul>
37 38 39	<ul> <li>If non-standardized extensions like !\$PRAGMA IGNORE_TKR are not available then S2 must be implemented with TYPE(*), DIMENSION().</li> </ul>
40 41 42 43	Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice argument may be implemented with an explicit interface using compiler directives, for example:
44	INTERFACE
45	SUBROUTINE MPI(buf,)
46	!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf
47 48	!\$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

 $\mathsf{MPI-3.0}$  (and later) compliant Fortran bindings are not only a property of the  $\mathsf{MPI}$  library itself, but rather a property of an  $\mathsf{MPI}$  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 TR 29113 [42] 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 628 for more details.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI\_SUBARRAYS\_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE..
- 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.
- 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.
- The Fortran compiler shall not provide TYPE(\*) unless the ASYNCHRONOUS attribute 46 protects MPI communication as described in TR 29113. Specifically, the TR 29113 47 must be implemented as a whole. 48

### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

<sup>1</sup> The following rules are required at least as long as the compiler does not provide the exten-<sup>2</sup> sion of the ASYNCHRONOUS attribute as part of TR 29113 and there is still one Fortran support <sup>3</sup> method with MPI\_ASYNC\_PROTECTS\_NONBLOCKING==.FALSE.. It is helpful when these <sup>4</sup> rules are observed, especially for backward compatibility of existing applications that use <sup>5</sup> the mpi module or the mpif.h include file. The rules are as follows:

- 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 639 and Section 17.1.8 on page 616, and DD on page 640) solve the problems described in Section 17.1.17 on page 634.
- The problems with temporary data movement (described in detail in Section 17.1.18 on page 641) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective IO) and the computation when overlapping communication and computation.
- Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 17.1.19 on page 643) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in calling MPI operations.

All of these rules are valid independently of whether the MPI routine interfaces in the mpi\_f08 and mpi modules are internally defined with an INTERFACE or CONTAINS construct, and with or without BIND(C), and also when mpif.h uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard if the MPI interfaces are defined without BIND(C). Additional compiler support may be necessary if BIND(C) is used. Some of these additional requirements are defined in the Fortran 2008 TR 29113 [42]. Some of these requirements for MPI-3.0 are beyond the scope of TR 29113. (*End of advice to implementors.*)

<sup>30</sup> Further requirements apply when the MPI library internally uses BIND(C) routine interfaces <sup>31</sup> (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 inidicating that these types may not be interoperable with an existing type in C. Some of these types are already valid in BIND(C) interfaces since Fortran 2003, some may be valid based on TR 29113 (e.g., CHARACTER\*(\*)).

- OPTIONAL dummy arguments are also valid within BIND(C) interfaces. This requirement is fulfilled if TR 29113 is fully supported by the compiler.
- <sup>44</sup><sub>45</sub> 17.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 17.1.17 on page 634, 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

7

8

9

10 11

12

13

14

15 16

17

18

19

20

24

25

26

27

28

29

32

33

34

35

36

37

38

39

40 41

42

$MPI_F_SYNC_REG(buf)$	
INOUT buf	initial address of buffer (choice)
<pre>MPI_F_sync_reg(buf) BIND(C)     TYPE(*), DIMENSION(),</pre>	ASYNCHRONOUS :: buf
<pre>MPI_F_SYNC_REG(buf)</pre>	

This routine is a no-operation. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

*Rationale.* This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is 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.)

Advice to users. If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI\_F\_SYNC\_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not to be called if MPI\_ASYNC\_PROTECTS\_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (*End of advice to users*.)

### 17.1.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI\_INTEGER, MPI\_REAL, MPI\_INT, MPI\_DOUBLE, etc., as well as the optional types MPI\_REAL4, MPI\_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

14

15

16

17 18

19

20

21 22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33 34

35

36

37

38

39

40 41 42

43 44

45

46

47

1 Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These  $\mathbf{2}$ types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL and 3 CHARACTER) with an optional integer KIND parameter that selects from among one or more 4 variants. The specific meaning of different KIND values themselves are implementation  $\mathbf{5}$ dependent and not specified by the language. Fortran provides the KIND selection functions 6 selected\_real\_kind for REAL and COMPLEX types, and selected\_int\_kind for INTEGER 7types that allow users to declare variables with a minimum precision or number of digits. 8 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX and 9 INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL 10 and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 11PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two 12declarations are equivalent:

double precision x real(KIND(0.0d0)) x

MPI provides two orthogonal methods to communicate using numeric intrinsic types. 16The first method (see the following section) can be used when variables have been de-17clared in a portable way — using default KIND or using KIND parameters obtained with the 18 selected\_int\_kind or selected\_real\_kind functions. With this method, MPI automati-19 cally selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion 20in heterogeneous environments. The second method (see Support for size-specific MPI 21Datatypes on page 622) gives the user complete control over communication by exposing 22 machine representations. 23

24

13

14

15

- 25
- 26 27

<sup>28</sup> Parameterized Datatypes with Specified Precision and Exponent Range

<sup>29</sup> <sub>30</sub> MPI provides named datatypes corresponding to standard Fortran 77 numeric types: MPI\_INTEGER, MPI\_COMPLEX, MPI\_REAL, MPI\_DOUBLE\_PRECISION and

<sup>31</sup> MPI\_DOUBLE\_COMPLEX. MPI automatically selects the correct data size and provides rep-<sup>32</sup> resentation conversion in heterogeneous environments. The mechanism described in this <sup>34</sup> section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables 35 are declared (perhaps indirectly) using selected\_real\_kind(p, r) to determine the KIND 36 parameter, where  $\mathbf{p}$  is decimal digits of precision and  $\mathbf{r}$  is an exponent range. Implicitly 37 MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 38 defined for each value of (p, r) supported by the compiler, including pairs for which one 39 value is unspecified. Attempting to access an element of the array with an index (p, r) not 40 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 41 datatypes. For integers, there is a similar implicit array related to selected\_int\_kind and 42indexed by the requested number of digits  $\mathbf{r}$ . Note that the predefined datatypes contained 43 in these implicit arrays are not the same as the named MPI datatypes MPI\_REAL, etc., but 44 a new set. 45

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.)

### Unofficial Draft for Comment Only

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.*)

#### 11 MPI\_TYPE\_CREATE\_F90\_REAL(p, r, newtype) 12IN precision, in decimal digits (integer) 13 р 14IN decimal exponent range (integer) r 15OUT newtype the requested MPI datatype (handle) 1617 int MPI\_Type\_create\_f90\_real(int p, int r, MPI\_Datatype \*newtype) 18 19 MPI\_Type\_create\_f90\_real(p, r, newtype, ierror) BIND(C) 20INTEGER, INTENT(IN) :: p, r 21TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23MPI\_TYPE\_CREATE\_F90\_REAL(P, R, NEWTYPE, IERROR) $^{24}$ INTEGER P, R, NEWTYPE, IERROR 25

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 621.

It is erroneous to supply values for  $\boldsymbol{p}$  and  $\boldsymbol{r}$  not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)			37
IN			
IIN	р	precision, in decimal digits (integer)	39
IN	r	decimal exponent range (integer)	40
OUT	newtype	the requested MPI datatype (handle)	41
			42
<pre>int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)</pre>			43
int Mr1_Type_create_190_comprex(int p, int 1, Mr1_Datatype *newtype)			44
MPI_Type_	<pre>create_f90_complex(p, r,</pre>	newtype, ierror) BIND(C)	45
INTEC	INTEGER, INTENT(IN) :: p, r		
TYPE(MPI_Datatype), INTENT(OUT) :: newtype			47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			

Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8 9 10

26

27

28

29

30

31

32

33

34

```
1
     MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
\mathbf{2}
          INTEGER P, R, NEWTYPE, IERROR
3
         This function returns a predefined MPI datatype that matches a
4
     COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from
5
     calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set
6
     to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to
7
     the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions
8
     on using the returned datatype with the "external32" data representation are given on page
9
     621.
10
         It is erroneous to supply values for p and r not supported by the compiler.
11
12
13
     MPI_TYPE_CREATE_F90_INTEGER(r, newtype)
14
       IN
                                             decimal exponent range, i.e., number of decimal digits
                 r
15
                                             (integer)
16
17
       OUT
                 newtype
                                             the requested MPI datatype (handle)
18
19
     int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
20
     MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
21
          INTEGER, INTENT(IN) :: r
22
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
23
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
^{24}
25
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
26
          INTEGER R, NEWTYPE, IERROR
27
     This function returns a predefined MPI datatype that matches a INTEGER variable of KIND
28
     selected_int_kind(r). Matching rules for datatypes created by this function are analo-
29
     gous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Re-
30
     strictions on using the returned datatype with the "external 32" data representation are
^{31}
     given on page 621.
32
         It is erroneous to supply a value for r that is not supported by the compiler.
33
         Example:
34
35
         integer
                        longtype, quadtype
36
         integer, parameter :: long = selected_int_kind(15)
37
         integer(long) ii(10)
38
        real(selected_real_kind(30)) x(10)
39
         call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
40
         call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
41
         . . .
42
43
         call MPI_SEND(ii, 10, longtype, ...)
44
         call MPI_SEND(x, 10, quadtype, ...)
45
46
                              The datatypes returned by the above functions are predefined
           Advice to users.
47
           datatypes. They cannot be freed; they do not need to be committed; they can be
48
           used with predefined reduction operations. There are two situations in which they
```

behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

- 1. MPI\_TYPE\_GET\_ENVELOPE returns special combiners that allow a program to retrieve the values of p and r.
- 2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI\_TYPE\_CREATE\_F90\_xxxx routines.

If a variable was declared specifying a non-default KIND value that was not obtained with selected\_real\_kind() or selected\_int\_kind(), the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section.

(End of advice to users.)

Advice to implementors. An application may often repeat a call to MPI\_TYPE\_CREATE\_F90\_xxxx with the same combination of (xxxx,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI\_TYPE\_CREATE\_F90\_xxxx and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (xxxx,p,r). (*End of advice to implementors.*)

*Rationale.* The MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2 on page 536) or user-defined (Section 13.5.3 on page 537) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.5.2 on page 536.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double" and "Double Extended" formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

```
The external32 representations of the datatypes returned by
MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules.
For MPI_TYPE_CREATE_F90_REAL:
```

```
if (p > 33) or (r > 4931) then external32 representation
is undefined
else if (p > 15) or (r > 307) then external32_size = 16
else if (p > 6) or (r > 37) then external32_size = 8
else external32_size = 4
```

 $\mathbf{2}$ 

 $^{24}$ 

```
1
     For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.
\mathbf{2}
     For MPI_TYPE_CREATE_F90_INTEGER:
3
         if
                  (r > 38) then external32 representation is undefined
4
         else if (r > 18) then external32_size =
                                                         16
5
         else if (r >
                         9) then
                                    external32_size =
                                                          8
6
         else if (r > 4) then external32_size =
                                                          4
7
         else if (r > 2) then
                                    external32_size =
                                                          2
8
         else
                                    external32_size = 1
9
10
     If the external 32 representation of a datatype is undefined, the result of using the datatype
11
     directly or indirectly (i.e., as part of another datatype or through a duplicated datatype)
12
     in operations that require the external 32 representation is undefined. These operations in-
13
     clude MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions,
14
     when the "external32" data representation is used. The ranges for which the external32
15
     representation is undefined are reserved for future standardization.
16
17
     Support for Size-specific MPI Datatypes
18
     MPI provides named datatypes corresponding to optional Fortran 77 numeric types that
19
     contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a
20
     mechanism that generalizes this model to support all Fortran numeric intrinsic types.
21
          We assume that for each typeclass (integer, real, complex) and each word size there is
22
     a unique machine representation. For every pair (typeclass, n) supported by a compiler,
23
     MPI must provide a named size-specific datatype. The name of this datatype is of the form
^{24}
     MPL_{TYPE>n in C and Fortran where TYPE> is one of REAL, INTEGER and COMPLEX,
25
     and \mathbf{n} is the length in bytes of the machine representation. This datatype locally matches
26
     all variables of type (typeclass, n). The list of names for such types includes:
27
28
     MPI_REAL4
29
     MPI_REAL8
30
     MPI_REAL16
^{31}
     MPI_COMPLEX8
32
     MPI_COMPLEX16
33
     MPI_COMPLEX32
34
     MPI_INTEGER1
35
     MPI_INTEGER2
36
     MPI_INTEGER4
37
     MPI_INTEGER8
38
     MPI_INTEGER16
39
40
     One datatype is required for each representation supported by the compiler. To be backward
^{41}
     compatible with the interpretation of these types in MPI-1, we assume that the nonstandard
42
     declarations REAL*n, INTEGER*n, always create a variable whose representation is of size n.
43
     These datatypes may also be used for variables declared with KIND=INT8/16/32/64 or
^{44}
     KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV intrinsic module. Note
45
     that the MPI datatypes and the REAL*n, INTEGER*n declarations count bytes whereas the
46
     Fortran KIND values count bits. All these datatypes are predefined.
47
          The following functions allow a user to obtain a size-specific MPI datatype for any
```

```
<sup>48</sup> intrinsic Fortran type.
```

MPI_SIZ	EOF(x, size)	1	1		
IN	х	a Fortran variable of numeric intrinsic type (choice)	2		
OUT	size		3 4		
			5		
MPI_Sizeof(x, size, ierror) BIND(C)					
TYPE(*), DIMENSION() :: x					
	EGER, INTENT(OUT)		8 9		
	EGER, OPTIONAL, I		9 .0		
	MPI_SIZEOF(X, SIZE, IERROR)				
•	pe> X	1	2		
TNL	EGER SIZE, IERROR	1	3		
This	s function returns t	he size in bytes of the machine representation of the given $1$	4		
variable.	It is a generic Forth	tan fourne and has a forman sinding only.	5		
Ad	vice to users This		.6		
			.8		
			9		
			0		
			1		
use	eful. (End of rational		2		
			3 24		
			:5		
			6		
IN	typeclass	generic type specifier (integer) 2	27		
IN	size	size, in bytes, of representation (integer) <sup>2</sup>	8		
OUT	datatype	datatype with correct type, size (handle)	9		
			1		
int MPI	_Type_match_size(	int typeclass, int size, MPI Datatype *datatype)	2		
MPI_Typ	e_match_size(type	eclass, size, datatype, ierror) BIND(C) 3	3		
			4		
	E(MPI_Datatype),		5		
INT	EGER, OPTIONAL, I		6		
MPI_TYP	E_MATCH_SIZE(TYPE	CLASS, STZE, DATATYPE, TERROR)	7 8		
INT	EGER TYPECLASS, S	STZE, DATATYPE, TERROR	9		
type	class is one of MPI_	TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and <sup>4</sup>	0		
21		,	1		
	<i>v</i> . 0	i local vallable of type (typeciass, size).	2		
		elefence (handle) to one of the predenned hanned datatypes, not	3		
-		t be need. Whit_TTTE_WATCH_SIZE can be used to obtain a	5		
-	size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF 4 in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find 4				
	=		7		

### **Unofficial Draft for Comment Only**

In addition, for variables of default kind the variable's size can be computed by a call to

```
624
                                                   CHAPTER 17. LANGUAGE BINDINGS
1
     MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not
\mathbf{2}
     supported by the compiler.
3
           Rationale. This is a convenience function. Without it, it can be tedious to find the
4
           correct named type. See note to implementors below. (End of rationale.)
5
6
           Advice to implementors. This function could be implemented as a series of tests.
7
8
           int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
9
           {
10
             switch(typeclass) {
11
                  case MPI_TYPECLASS_REAL: switch(size) {
12
                    case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
13
                    case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
14
                    default: error(...);
15
                  }
16
                  case MPI_TYPECLASS_INTEGER: switch(size) {
17
                     case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
18
                     case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
19
                     default: error(...);
20
                  }
21
                 ... etc. ...
22
              }
23
24
              return MPI_SUCCESS;
25
           }
26
27
           (End of advice to implementors.)
28
29
     Communication With Size-specific Types
30
^{31}
     The usual type matching rules apply to size-specific datatypes: a value sent with datatype
32
     MPI_{TYPE>n} can be received with this same datatype on another process. Most modern
33
     computers use 2's complement for integers and IEEE format for floating point. Thus, com-
34
     munication using these size-specific datatypes will not entail loss of precision or truncation
35
     errors.
36
37
           Advice to users. Care is required when communicating in a heterogeneous environ-
38
           ment. Consider the following code:
39
40
           real(selected_real_kind(5)) x(100)
41
           call MPI_SIZEOF(x, size, ierror)
42
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
43
           if (myrank .eq. 0) then
44
                ... initialize x ...
45
               call MPI_SEND(x, xtype, 100, 1, ...)
46
           else if (myrank .eq. 1) then
47
               call MPI_RECV(x, xtype, 100, 0, ...)
48
           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 3 environment. To communicate in a heterogeneous environment, there are at least four 4 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 5is to use selected\_real\_kind or selected\_int\_kind and with the functions of the 6  $\overline{7}$ 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 9 result in an 8-byte representation). The fourth is to carefully check representation 10 size before communication. This may require explicit conversion to a variable of size 11 that can be communicated and handshaking between sender and receiver to agree on 12a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
16
real(selected_real_kind(5)) x(100)
                                                                               17
call MPI_SIZEOF(x, size, ierror)
                                                                               18
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
                                                                               19
                                                                               20
if (myrank .eq. 0) then
                                                                               21
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                               &
                                                                               22
                                                               &
                       MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                                               23
                       MPI_INFO_NULL, fh, ierror)
                                                                               ^{24}
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                               25
                           MPI_INFO_NULL, ierror)
                                                                               26
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                               27
   call MPI_FILE_CLOSE(fh, ierror)
                                                                               28
endif
                                                                               29
                                                                               30
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
                                                                               31
                                                                               32
if (myrank .eq. 1) then
                                                                               33
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY, &
                                                                               34
                  MPI_INFO_NULL, fh, ierror)
                                                                               35
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                               36
                           MPI_INFO_NULL, ierror)
                                                                               37
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                               38
   call MPI_FILE_CLOSE(fh, ierror)
                                                                               39
endif
                                                                               40
                                                                               41
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (End of advice to users.)

#### 17.1.10Problems With Fortran Bindings for MPI

47This section discusses a number of problems that may arise when using MPI in a Fortran 48 program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

8

13

14

15

42

43

44 45

<sup>1</sup> does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TR 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

- An MPI subroutine with a choice argument may be called with different argument types. When using the mpi\_f08 module together with a compiler that supports Fortran 2008 + TR 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- <sup>18</sup>
   3. Nonblocking and split-collective MPI routines assume that actual arguments are passed <sup>19</sup> by address or descriptor and that arguments and the associated data are not copied <sup>20</sup> on entrance to or exit from the subroutine. This problem is solved with the use of the <sup>21</sup> ASYNCHRONOUS attribute.
  - 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TR 29113.
  - 5. Several named "constants," such as MPI\_BOTTOM, MPI\_IN\_PLACE, MPI\_STATUS\_IGNORE, MPI\_STATUSES\_IGNORE, MPI\_ERRCODES\_IGNORE, MPI\_UNWEIGHTED, MPI\_WEIGHTS\_EMPTY, MPI\_ARGV\_NULL, and MPI\_ARGVS\_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 15 for more information.

6. The memory allocation routine MPI\_ALLOC\_MEM can't be usefully used in Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 - MPI-2.2. In Fortran 2003, TYPE(C\_PTR) entities were added, which allow a standard-conforming implementation of the semantics of MPI\_ALLOC\_MEM. In MPI-3.0 and later, MPI\_ALLOC\_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The mpi\_f08 module only supports TYPE(C\_PTR) pointers.

- Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
  - MPI identifiers exceed 6 characters.
  - MPI identifiers may contain underscores after the first character.
- MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.

# Unofficial Draft for Comment Only

9

10

11

12

13 14

15

16

17

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

• 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 615.

# 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 611). In C, the use of void\* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TR 29113 (and later) together with the mpi\_f08 module, the problem is avoided by declaring choice arguments with TYPE(\*), DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE mpif.h, the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. Using the mpi\_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi\_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI\_CART\_CREATE are declared as assumed size arrays INTEGER :: DIMS(\*) and LOGICAL :: PERIODS(\*).

```
USE mpi_f08 ! or USE mpi
INTEGER size
CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
```

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

1 Although this is a non-conforming MPI call, compiler warnings are not expected (but may  $\mathbf{2}$ occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit 3 interfaces. 4 517.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets 6 Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g., 7 8 REAL a(100,100,100) 9 CALL MPI\_Send( a(11:17, 12:99:3, 1:100), 7\*30\*100, MPI\_REAL, ...) 10 11The handling of subscript triplets depends on the value of the constant 12MPI\_SUBARRAYS\_SUPPORTED: 13 • If MPI\_SUBARRAYS\_SUPPORTED equals .TRUE.: 14Choice buffer arguments are declared as TYPE(\*), DIMENSION(...). For example, 15consider the following code fragment: 1617 18 REAL s(100), r(100) 19 CALL MPI\_Isend(s(1:100:5), 3, MPI\_REAL, ..., rq, ierror) CALL MPI\_Wait(rq, status, ierror) 20CALL MPI\_Irecv(r(1:100:5), 3, MPI\_REAL, ..., rq, ierror) 21CALL MPI\_Wait(rq, status, ierror) 222324In this case, the individual elements s(1), s(6), and s(11) are sent between the start of MPI\_ISEND and the end of MPI\_WAIT even though the compiled code will not copy 2526s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code will pass a descriptor to MPI\_ISEND that allows MPI to operate directly on s(1), s(6), 27s(11), ..., s(96). The called MPI\_ISEND routine will take only the first three of these 28elements due to the type signature "3, MPI\_REAL". 2930 All nonblocking MPI functions (e.g., MPI\_ISEND, MPI\_PUT, 31MPI\_FILE\_WRITE\_ALL\_BEGIN) behave as if the user-specified elements of choice 32 buffers are copied to a contiguous scratch buffer in the MPI runtime environment. 33 All datatype descriptions (in the example above, "3, MPI\_REAL") read and store 34 data from and to this virtual contiguous scratch buffer. Displacements in MPI de-35rived datatypes are relative to the beginning of this virtual contiguous scratch buffer. 36 Upon completion of a nonblocking receive operation (e.g., when MPI\_WAIT on a cor-37 responding MPI\_Request returns), it is as if the received data has been copied from 38 the virtual contiguous scratch buffer back to the non-contiguous application buffer. 39 In the example above, r(1), r(6), and r(11) are guaranteed to be defined with the 40 received data when MPI\_WAIT returns. 41 42Advice to implementors. The Fortran descriptor for TYPE(\*), DIMENSION(...) 43 arguments contains enough information that, if desired, the MPI library can make 44a real contiguous copy of non-contiguous user buffers when the nonblocking op-45eration is started, and released this buffer not before the nonblocking commin-46cation has completed (e.g., in an MPI wait routine). Efficient implementations 47 may avoid such additional memory-to-memory data copying. (End of advice to 48

*implementors.*)

*Rationale.* If MPI\_SUBARRAYS\_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside of the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (*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, ...)
```

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

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

<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.

	630	CHAPTER 17. LANGUAGE BINDINGS
$\frac{1}{2}$		<pre>name ( [:,] [<subscript>]:[<subscript>] [,<subscript>] )</subscript></subscript></subscript></pre>
2 3 4 5 6 7		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
8		A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
10 11		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.
12 13 14 15		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
16 17 18 19 20		real :: a call user1(a,rq) call MPI_WAIT(rq,status,ierr) write (*,*) a
21 22 23 24		<pre>subroutine user1(buf,request) call MPI_IRECV(buf,,request,) end</pre>
25 26 27		If a is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter a itself.
28 29 30 31 32		Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., $A(1:n:2)$ ), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.
33 34		If a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this must be employed.
35 36 37 38 39 40 41 42 43		If a compiler makes copies in the calling procedure of arguments that are explicit- shape 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.
44 45 46 47 48		13 Problems Due to Data Copying and Sequence Association with Vector Subscripts can arrays with <b>vector</b> subscripts describe subarrays containing a possibly irregular f 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.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
    integer :: i
    real :: x
    double precision :: d
    logical :: 1
end type mytype
type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%l, disp(4), ierr)
base = disp(1)
```

Unofficial Draft for Comment Only

 $45 \\ 46$ 

```
1
          disp(1) = disp(1) - base
\mathbf{2}
          disp(2) = disp(2) - base
3
          disp(3) = disp(3) - base
4
          disp(4) = disp(4) - base
5
6
          blocklen(1) = 1
7
          blocklen(2) = 1
8
          blocklen(3) = 1
9
          blocklen(4) = 1
10
11
          type(1) = MPI_INTEGER
12
          type(2) = MPI_REAL
13
          type(3) = MPI_DOUBLE_PRECISION
14
          type(4) = MPI_LOGICAL
15
16
          call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
17
          call MPI_TYPE_COMMIT(newtype, ierr)
18
19
20
21
          call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
22
     ! or
23
          call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
24
          ! expects that base == address(foo%i) == address(foo)
25
26
          call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
27
          call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
28
          extent = disp(2) - disp(1)
29
          1b = 0
30
          call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
31
          call MPI_TYPE_COMMIT(newarrtype, ierr)
32
33
          call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
34
          Using the derived type variable foo instead of its first basic type element foo%i may
35
     be impossible if the MPI library implements choice buffer arguments through overloading
36
     instead of using TYPE(*), DIMENSION(...), or through a non-standardized extensions such
37
     as !$PRAGMA IGNORE_TKR; see Section 17.1.6 on page 611.
38
          To use a derived type in an array requires a correct extent of the datatype handle to
39
     take care of the alignment rules applied by the compiler. These alignment rules may imply
40
     that there are gaps between the elements of a derived type, and also between the array
41
     elements. The extent of an interoperable derived type (i.e., defined with BIND(C)) and a
42
     SEQUENCE derived type with the same content may be different because C and Fortran may
43
     apply different alignment rules. As recommended in the advice to users in Section 4.1 on
44
     page 83, one should add an additional fifth structure element with one numerical storage
45
     unit at the end of this structure to force in most cases that the array of structures is
46
     contiguous. Even with such an additional element, one should keep this resizing due to the
47
     special alignment rules that can be used by the compiler for structures, as also mentioned
48
```

in this advice.

Using the extended semantics defined in TR 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 as it may fit for such types. Sending the structure foo should then also be performed by providing it (and not foo%i) as actual argument for MPI\_Send.

### 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*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 17.1.17 on page 634.
- Temporary data movement and temporary memory modifications; see Section 17.1.18 on page 641.
- Permanent data movement (e.g., through garbage collection); see Section 17.1.19 on page 643.

Table 17.1 shows in which usage areas the optimization problems may only occur. The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" to "VOLATILE" on pages 636-637,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 17.1.7 on page 615.

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

1	Optimization .			ay cause llowing u	-	
3			Nonbl.	1-sided	. –	Bottom
4 5	Code movemen and register op		yes	yes	no	yes
6	Temporary dat		yes	yes	yes	no
7	Permanent dat	a movement	yes	yes	yes	yes
8 9 10 11	Table 17.1: Occurrence	e of Fortran oj	otimizatio	on proble	ms in se	everal usage areas
12	17.1.17 Problems with Co	de Movement	and Re	gister Op	timizat	ion
13 14	Nonblocking operations					
15 16 17 18 19 20	If a variable is local to a For compiler will assume that it argument of the call. In the to save and restore certain re- held a valid copy of such a v	cannot be mod most commo egisters. Thus	dified by n linkage , the opti	a called s conventi mizer wil	ubroutin on, the ll assum	ne unless it is an actual subroutine is expected he that a register which
21 22	Example 17.1 Fortran 90 r	egister optimi	ization –	extreme.		
23	Source	compiled as		OI	compil	led as
24 25 26 27 28	<pre>REAL :: buf, b1 call MPI_IRECV(buf,req) call MPI_WAIT(req,) b1 = buf</pre>	REAL :: buf call MPI_IRJ register = 1 call MPI_WA b1 = registe	ECV(buf,. buf IT(req,	R .req) c b	EAL :: all MPI 1 = buf	buf, b1 [_IRECV(buf,req)
29 30 31 32 33 34 35 36	Example 17.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_IRECV and the finish of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has returned, and may schedule the load of buf earlier than typed in the source. The compiler has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as illustrated in the rightmost column.					
37 38	Example 17.2 Similar example	nple with $MP$	I_ISEND			
39	Source	compiled as			-	ossible MPI-internal n sequence
40 41 42 43 44	<pre>REAL :: buf, copy buf = val call MPI_ISEND(buf,req) copy = buf</pre>	REAL :: buf buf = val call MPI_ISI copy= buf buf = val_ov	END(buf,.	R b .req) a c b	EAL :: uf = va ddr = & opy= bu uf = va	buf, copy ll buf f l_overwrite
45 46 47 48	<pre>call MPI_WAIT(req,) buf = val_overwrite</pre>	call MPI_WA	- · ( · e q , · ·	, 5		ldr) ! within MPI_WAIT

47

48

1 Due to valid compiler code movement optimizations in Example 17.2, the content of  $\mathbf{2}$ buf may already be overwritten by the compiler when the content of buf is sent. The 3 code movement is permitted because the compiler cannot detect a possible access to buf 4 in MPI\_WAIT (or in a second thread between the start of MPI\_ISEND and the end of 5MPI\_WAIT). Such register optimization is based on moving code; here, the access to buf was moved 6  $\overline{7}$ from after MPI\_WAIT to before MPI\_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined. 8 9 This register optimization / code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ...\_BEGIN 10 11and ...\_END calls, the same buffer has to be provided as an actual argument. The register 12optimization / code movement problem for MPI\_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication or parallel file I/O operation. 13 1415One-sided communication 16An example with instruction reordering due to register optimization can be found in Sec-17 tion 11.7.4 on page 464. 18 19MPI\_BOTTOM and combining independent variables in datatypes 2021This section is only relevant if the MPI program uses a buffer argument to an 22 MPI\_SEND, MPI\_RECV etc., which hides the actual variables involved. MPI\_BOTTOM with 23an MPI\_Datatype containing absolute addresses is one example. Creating a datatype which  $^{24}$ uses one variable as an anchor and brings along others by using MPI\_GET\_ADDRESS to 25determine their offsets from the anchor is another. The anchor variable would be the only 26one referenced in the call. Also attention must be paid if MPI operations are used that run 27in parallel with the user's application. 28Example 17.3 shows what Fortran compilers are allowed to do. 2930 **Example 17.3** Fortran 90 register optimization. 3132 This source ... can be compiled as: 33 call MPI\_GET\_ADDRESS(buf, bufaddr, call MPI\_GET\_ADDRESS(buf,...) 34 ierror) 35call MPI\_TYPE\_CREATE\_STRUCT(1,1, call MPI\_TYPE\_CREATE\_STRUCT(...) 36 bufaddr, 37 MPI\_REAL, type, ierror) 38 call MPI\_TYPE\_COMMIT(...) call MPI\_TYPE\_COMMIT(type,ierror) 39 val\_old = buf register = buf 40 val\_old = register 41 call MPI\_RECV(MPI\_BOTTOM,...) call MPI\_RECV(MPI\_BOTTOM,1,type,...) 42val\_new = buf val\_new = register 43 4445In Example 17.3, the compiler does not invalidate the register because it cannot see 46that MPI\_RECV changes the value of buf. The access to buf is hidden by the use of

#### Unofficial Draft for Comment Only

MPI\_GET\_ADDRESS and MPI\_BOTTOM.

Example 17.4 Similar example with MPI\_SEND  $\mathbf{2}$ 3 This source ... can be compiled as: 4 ! buf contains val\_old ! buf contains val\_old 5buf = val\_new 6 call MPI\_SEND(MPI\_BOTTOM,1,type,...) call MPI\_SEND(...) 7 ! with buf as a displacement in type ! i.e. val\_old is sent 8 Ţ 9 ! buf=val\_new is moved to here 10 ! and detected as dead code 11 ! and therefore removed 12! 13 buf = val\_overwrite buf = val\_overwrite 14

15 16

17

18

19

20

21

1

In Example 17.4, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI\_SEND statement is interfering because the load access to buf is hidden by the usage of MPI\_BOTTOM.

#### 22 23 Solutions

The following sections show in detail how the problems with code movement and register optimization can be solved in a portable way. 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 datatype handles that combine several variables are used:

30 31

32

33

34

- 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.
- 35 36 37
- Use of the Fortran VOLATILE attribute.

38 Each of these methods solves the problems of code movement and register optimization, 39 but may involve different degrees of performance impact, and may not be usable in every 40 application context. These methods may not be guaranteed by the Fortran standard, but 41 they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated 42compiler suite according to the requirements listed in Section 17.1.7 on page 615. The 43 methods may have different impact on performance. MPI\_F\_SYNC\_REG may have low 44 impact, module data and the ASYNCHRONOUS attribute low through medium, and the 45VOLATILE attribute may have the most negative impact on performance. Note that there is 46 one attribute that cannot be used for this purpose: the Fortran TARGET attribute does not 47solve code movement problems in MPI applications. 48

```
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
```

# The Fortran ASYNCHRONOUS attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping 37 unit (or BLOCK) tells the compiler that any statement in the scoping unit may be executed 38 while the buffer is affected by a pending asynchronous Fortran input/output operation (since 39 Fortran 2003) or by an asynchronous communication (TR 29113 extension). Without the 40 extensions specified in TR 29113, a Fortran compiler may totally ignore this attribute if the 41 Fortran compiler implements asynchronous Fortran input/output operations with blocking 42I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through 43 code movements across routine calls, and the buffer itself from temporary and permanent 44data movements. If the choice buffer dummy argument of a nonblocking MPI routine is 45declared with ASYNCHRONOUS (which is mandatory for the mpi\_f08 module, with allowable 46 exceptions listed in Section 17.1.6 on page 611), then the compiler has to guarantee call by 47reference and should report a compile-time error if call by reference is impossible, e.g., if 48

# Unofficial Draft for Comment Only

1

2 3

4

5

6

7

8

9

10

11

12

13 14

15

16

17

18

19

20

21 22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33 34 35

vector subscripts are used. The MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if
 both the protection of the actual buffer argument through ASYNCHRONOUS according to the
 TR 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in
 the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set
 to .FALSE..

The ASYNCHRONOUS attribute has some restrictions. The TR 29113 defines (in the PDTR N1869):

"Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent. Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 17.5 Case (a) on page 637, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication between the MPI\_I... routines and MPI\_Waitall. Case (a) works fine because the read accesses to b occur after the communication completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to a pending communication affector while input communication (i.e., the two MPI\_Irecv calls) 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 disjunct subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 17.9 on page 645.

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.

 $44 \\ 45$ 

39

40

41

42

43

638

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

- -10
- 46
- 47
- 48

# 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 616.

• 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
can be solved with	can be solved with
<pre>call MPI_IRECV(buf,req)</pre>	buf = val
	<pre>call MPI_ISEND(buf,req)</pre>
	copy = buf
<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
b1 = buf	<pre>buf = val_overwrite</pre>

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
can be solved with	can be solved with
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)

The first call to MPI\_F\_SYNC\_REG(buf) is needed to finish all load and store references to buf prior to MPI\_RECV/MPI\_SEND; the second call is needed to assure that the subsequent access to buf are not moved before MPI\_RECV/SEND.

• In the example in Section 11.7.4 on page 464, 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 after the second fence, a call to MPI\_F\_SYNC\_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI\_WIN\_FENCE. Using MPI\_GET instead of MPI\_PUT, the same calls to MPI\_F\_SYNC\_REG are necessary.

Source of Process 1	Source of Process 2

 $^{24}$ 

1 bbbb = 777buff = 999 $\mathbf{2}$ call MPI\_F\_SYNC\_REG(buff) 3 call MPI\_WIN\_FENCE call MPI\_WIN\_FENCE 4 call MPI\_PUT(bbbb 5into buff of process 2) 6 7 call MPI\_WIN\_FENCE call MPI\_WIN\_FENCE 8 call MPI\_F\_SYNC\_REG(bbbb) call MPI\_F\_SYNC\_REG(buff) 9 ccc = buff10 11 • The temporary memory modification problem, i.e., Example 17.6 on page 641, can 12**not** be solved with this method. 13 14A user defined routine instead of MPI\_F\_SYNC\_REG 1516Instead of MPI\_F\_SYNC\_REG, one can also use a user defined external subroutine, which 17is separately compiled: 18 19subroutine DD(buf) 20integer buf 21end 22 Note that if the intent is declared in an explicit interface for the external subroutine, 23it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler  $^{24}$ does not know this and has to assume that the buffer may be altered. For example, a call 25to MPI\_RECV with MPI\_BOTTOM as buffer might be replaced by 2627call DD(buf) 28call MPI\_RECV(MPI\_BOTTOM,...) 29 call DD(buf) 30 Such a user-defined routine was introduced in MPI-2.0 and is still included here to document  $^{31}$ such usage in existing application programs although new applications should prefer 32 MPI\_F\_SYNC\_REG or one of the other posibilities. In an existing application, calls to such a 33 34user-written routine should be substituted by a call to MPI\_F\_SYNC\_REG because the userwritten routine may not be implemented according to the rules specified in Section 17.1.7 35 on page 615. 36 37 38 Module variables and COMMON blocks 39 An alternative to the already mentioned methods is to put the buffer or variable into a 40 module or a common block and access it through a USE or COMMON statement in each scope 41 where it is referenced, defined or appears as an actual argument in a call to an MPI routine. 42The compiler will then have to assume that the MPI procedure may alter the buffer or 43 variable, provided that the compiler cannot infer that the MPI procedure does not reference 44 the module or common block. 45 46• This method solves problems of instruction reordering, code movement, and register 47optimization related to nonblocking and one-sided communication, or related to the 48 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, but it may inhibit optimization of any code containing references or definitions of the buffer or variable.

### 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 TR 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (End of rationale.)

#### Temporary Data Movement and Temporary Memory Modification 17.1.18

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, Case (b) on page 637. Example 17.6 on page 641 shows a possibility that could be problematic.

#### Example 17.6 Overlapping Communication and Computation.

USE mpi_f08 REAL :: buf(100,100) CALL MDL Imagy(buf(1,1:100) = mag )
CALL MPI_Irecv(buf(1,1:100),req,)
DO j=1,100
DO i=2,100
buf(i,j)=
END DO
END DO
CALL MPI_Wait(req,)

In the compiler-generated, possible optimization in Example 17.7, 41 buf(100,100) from Example 17.6 is equivalenced with the 1-dimensional array 42buf\_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer. 4344When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

47Example 17.8 shows a second possible optimization. The whole array is temporarily 48 moved to local\_buf. When storing local\_buf back to the original location buf, then this

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4 5

> 6 7

> 8

9 10

11

12

13

14

15

1617

18

19

2021

2223

 $^{24}$ 

25

2627

40

```
1
     Example 17.7 The compiler may substitute the nested loops through loop fusion.
\mathbf{2}
3
     REAL :: buf(100,100), buf_1dim(10000)
4
     EQUIVALENCE (buf(1,1), buf_1dim(1))
5
     CALL MPI_Irecv(buf(1,1:100),...req,...)
6
     tmp(1:100) = buf(1,1:100)
7
     DO j=1,10000
8
       buf_1dim(h)=...
9
     END DO
10
     buf(1,1:100) = tmp(1:100)
11
     CALL MPI_Wait(req,...)
12
13
14
     Example 17.8 Another optimization is based on the usage of a separate memory storage
15
     area, e.g., in a GPU.
16
17
     REAL :: buf(100,100), local_buf(100,100)
18
     CALL MPI_Irecv(buf(1,1:100),...req,...)
19
     local_buf = buf
     DO j=1,100
20
21
       DO i=2,100
22
          local_buf(i,j)=....
23
       END DO
^{24}
     END DO
25
     buf = local_buf ! may overwrite asynchronously received
26
                        ! data in buf(1,1:100)
27
     CALL MPI_Wait(req,...)
28
29
30
     includes also an overwriting of the receive buffer part buf(1,1:100), i.e., this storing back
31
     may overwrite the asynchronously received data.
32
          Note, that this problem may also occur:
33
         • With the local buffer at the origin process, between an RMA communication call and
34
           the ensuing synchronization call; see Chapter 11 on page 403.
35
36
         • With the window buffer at the target process between two ensuing RMA synchroniza-
37
           tion calls.
38
         • With the local buffer in MPI parallel file I/O split collective operations with between
39
           the ..._BEGIN and ..._END calls; see Section 13.4.5 on page 526.
40
41
          As already mentioned in subsection The Fortran ASYNCHRONOUS attribute on page 637
42
     in Section 17.1.17 on page 634, the ASYNCHRONOUS attribute can prevent compiler optimiza-
43
     tion with temporary data movement, but only if the receive buffer and the numerical read
44
     accesses are separated into different variables, as shown in Example 17.9 on page 645 and
45
     in Example 17.10 on page 646.
46
          Note also that the methods
47
48
         • calling MPI_F_SYNC_REG (or such a user-defined routine),
```

- using module variables and COMMON blocks, and
- the TARGET attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the numerical code shown in Example 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 needed to **prevent** the problem. When overlapping communication and computation, the nonblocking communication (or nonblocking or split collective IO) and the computation should be executed **on different sets of variables**. 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 restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature is available in a later version of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 17.9 on page 645 (which is a solution for the problem shown in Example 17.5 on page 637) and in Example 17.10 on page 646 (which is a solution for the problem shown in Example 17.8 on page 642), the array is split into inner and halo part and both disjunct parts are passed to a subroutine separated\_sections. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the calculation on the elements where inner+halo is needed. Note that the halo and the inner area are strided arrays. Those can be used in non-blocking communication only with a TR 29113 based MPI library.

#### 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. Automatic garbage collection implementation is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with MPI\_BOTTOM.
- Nonblocking MPI operations (communication, one-sided, I/O) if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 17.1.7 on page 615.

#### Unofficial Draft for Comment Only

> $\mathbf{5}$

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 

#### 17.1.20 Comparison with C $\mathbf{2}$ In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the & operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while the nonblocking operations are pending.

**Example 17.9** Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
10
USE mpi_f08
                                                                                  11
REAL :: b(0:101) ! elements 0 and 101 are halo cells
                                                                                  12
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                  13
INTEGER :: i
                                                                                  14
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
                                                                                  15
i=1 ! compute leftmost element
                                                                                  16
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                  17
i=100 ! compute rightmost element
                                                                                  18
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                  19
END
                                                                                  20
                                                                                  21
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
                                                                                  22
USE mpi_f08
                                                                                  23
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
                                                                                  24
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                  25
TYPE(MPI_Request) :: req(4)
                                                                                  26
INTEGER :: left, right, i
                                                                                  27
CALL MPI_Cart_shift(...,left,right,...)
                                                                                  28
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
                                                                                  29
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
                                                                                  30
! b_lefthalo and b_righthalo is written asynchronously.
                                                                                  31
! There is no other concurrent access to b_lefthalo and b_righthalo.
                                                                                  32
CALL MPI_Isend(b_inner( 1), ..., left, ..., req(3), ...)
                                                                                  33
CALL MPI_Isend(b_inner(100), ..., right, ..., req(4), ...)
                                                                                  34
                                                                                  35
DO i=2,99 ! compute only elements for which halo data is not needed
                                                                                  36
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
                                                                                  37
  ! b_inner is read and send at the same time.
                                                                                  38
  ! This is allowed based on the rules for ASYNCHRONOUS.
                                                                                  39
END DO
                                                                                  40
CALL MPI_Waitall(4,req,...)
                                                                                  41
END SUBROUTINE
                                                                                  42
                                                                                  43
```

1 2

4 5 6

8

9

```
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 applications 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

# Unofficial Draft for Comment Only

1 2

3 4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15 16

17

18

19

20 21

22

23

 $^{24}$ 

25 26

27 28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41 42

43 44

45

 $46 \\ 47$ 

1 executing processes. Use of the Fortran version of MPI\_INIT to initialize MPI may 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 8 caller or by the processes killed. The MPI environment is initialized in the same manner for all languages by 9 MPI\_INIT. E.g., MPI\_COMM\_WORLD carries the same information regardless of language: 10 same processes, same environmental attributes, same error handlers. 11 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 28in the following conversion functions. With the Fortran mpi\_f08 module, a Fortran handle 29is a BIND(C) derived type that contains an INTEGER field 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 2.6.433 on page 19. 34 MPI\_Comm MPI\_Comm\_f2c(MPI\_Fint comm) 35 36 If comm is a valid Fortran handle to a communicator, then MPI\_Comm\_f2c returns a 37 valid C handle to that same communicator; if  $comm = MPI_COMM_NULL$  (Fortran value), 38 then MPI\_Comm\_f2c returns a null C handle; if comm is an invalid Fortran handle, then 39 MPI\_Comm\_f2c returns an invalid C handle. 40 MPI\_Fint MPI\_Comm\_c2f(MPI\_Comm comm) 41 42The function MPI\_Comm\_c2f translates a C communicator handle into a Fortran handle 43 to the same communicator; it maps a null handle into a null handle and an invalid handle 44into an invalid handle. 45Similar functions are provided for the other types of opaque objects. 46 MPI\_Datatype MPI\_Type\_f2c(MPI\_Fint datatype) 4748MPI\_Fint MPI\_Type\_c2f(MPI\_Datatype datatype)

MPI_Group MPI_Group_f2c(MPI_Fint group)	1
MPI_Fint MPI_Group_c2f(MPI_Group group)	2 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 10
MPI_Win MPI_Win_f2c(MPI_Fint win)	11
MPI_Fint MPI_Win_c2f(MPI_Win win)	12 13
MPI_Op MPI_Op_f2c(MPI_Fint op)	14
MPI_Fint MPI_Op_c2f(MPI_Op op)	15
MPI_Info MPI_Info_f2c(MPI_Fint info)	16 17
MPI_Fint MPI_Info_c2f(MPI_Info info)	18
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	19
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	20 21
MPI_Message MPI_Message_f2c(MPI_Fint message)	22
	23 24
MPI_Fint MPI_Message_c2f(MPI_Message message)	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	29
interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.	30
	31 32
! FORTRAN PROCEDURE SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)	33
INTEGER :: DATATYPE, IERR	34
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	35 36
RETURN END	30 37
END	38
/* C wrapper */	39 40
<pre>void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)</pre>	40
{	42
MPI_Datatype datatype;	43
<pre>datatype = MPI_Type_f2c( *f_handle);</pre>	44 45
<pre>*ierr = (MPI_Fint)MPI_Type_commit( &amp;datatype);</pre>	46
<pre>*f_handle = MPI_Type_c2f(datatype);</pre>	47
return;	48

1

2

3

4

56

7

8

9

10

11

12

13

14

15

1617

18

21

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

19 The following two procedures are provided in C to convert from a Fortran (with the mpi 20module 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 22 is hidden. That is, no status information is lost in the conversion. 23

 $^{24}$ 25

26

27

28

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. 29The C status has the same source, tag and error code values as the Fortran status, 30 and returns the same answers when queried for count, elements, and cancellation. The  $^{31}$ conversion function may be called with a Fortran status argument that has an undefined 32 error field, in which case the value of the error field in the C status argument is undefined. 33

Two global variables of type MPI\_Fint\*, MPI\_F\_STATUS\_IGNORE and 34MPI\_F\_STATUSES\_IGNORE are declared in mpi.h. They can be used to test, in C, whether 35 f\_status is the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE defined in 36 the mpi module or mpif.h. These are global variables, not C constant expressions and 37 cannot be used in places where C requires constant expressions. Their value is defined only 38 between the calls to MPI\_INIT and MPI\_FINALIZE and should not be changed by user code. 39 To do the conversion in the other direction, we have the following: 40

int MPI\_Status\_c2f(const MPI\_Status \*c\_status, MPI\_Fint \*f\_status) 41

42This call converts a C status into a Fortran status, and has a behavior similar to 43MPI\_Status\_f2c. That is, the value of c\_status must not be either MPI\_STATUS\_IGNORE or 44MPI\_STATUSES\_IGNORE. 45

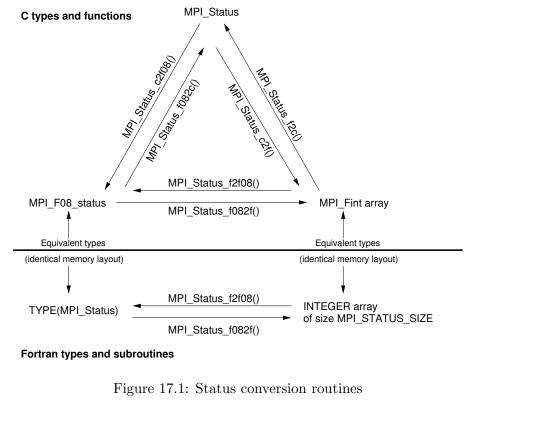
46 Advice to users. There exists no separate conversion function for arrays of statuses, 47 since one can simply loop through the array, converting each status with the routines 48 in Fig. 17.1 on page 651. (End of advice to users.)

# Unofficial Draft for Comment Only

}

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.



# 

This C routine converts a Fortran mpi\_f08 TYPE(MPI\_Status) into a C MPI\_Status.

# 

This C routine converts a C MPI\_Status into a Fortran mpi\_f08 TYPE(MPI\_Status). <sup>42</sup> Two global variables of type MPI\_F08\_status\*, MPI\_F08\_STATUS\_IGNORE and <sup>43</sup> MPI\_F08\_STATUSES\_IGNORE are declared in mpi.h. They can be used to test, in C, whether <sup>44</sup> f\_status is the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE defined in <sup>45</sup> the mpi\_f08 module. These are global variables, not C constant expressions and cannot be <sup>46</sup> used in places where C requires constant expressions. Their value is defined only between <sup>47</sup> the calls to MPI\_INIT and MPI\_FINALIZE and should not be changed by user code. <sup>48</sup>

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

```
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
44
     information, and have the same meaning in both languages. The mechanism described
45
     in the previous section can be used to pass references to MPI objects from language to
46
     language. An object created in one language can be accessed, modified or freed in another
```

48

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 17.2.9, page 659).

#### 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];
   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;
   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.
                                                                */
```

**Unofficial Draft for Comment Only** 

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

1213

14

1516

17

18

19

26

31

32

33

40

41

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.*)

<sup>23</sup> 24 Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are associated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language and language support method), and is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI\_COMM\_NULL\_COPY\_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI\_COMM\_CREATE\_KEYVAL); see also the advice to users on page 270. (End of advice to users.)

}

#### Error Handlers

Advice to implementors. Error handlers, have, in C, a "stdargs" argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

### **Reduce Operations**

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 define attributes arguments to be of type void\* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C 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 Fortran function MPI\_ATTR\_GET will return the least significant part of the attribute word; the 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, 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

#### **Unofficial Draft for Comment Only**

 $^{24}$ 

1 is accessed from C, then MPI\_xxx\_get\_attr will return the address of (a pointer to) the  $\mathbf{2}$ integer valued attribute, which is a pointer to MPI\_Aint if the attribute was stored with 3 Fortran MPI\_xxx\_SET\_ATTR, and a pointer to int if it was stored with the deprecated 4 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  $\mathbf{5}$ 6 conversion. This conversion is lossless if new style attribute functions are used, and an 7integer of kind MPI\_ADDRESS\_KIND is returned. The conversion may cause truncation if 8 deprecated attribute functions are used. In C, the deprecated routines MPI\_Attr\_put and 9 MPI\_Attr\_get behave identical to MPI\_Comm\_set\_attr and MPI\_Comm\_get\_attr. 10 Example 17.13 11 A. Setting an attribute value in C 1213 int set\_val = 3; 14struct foo set\_struct; 1516/\* Set a value that is a pointer to an int \*/ 1718 MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval1, &set\_val); 19 /\* Set a value that is a pointer to a struct \*/ 20MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval2, &set\_struct); 21/\* Set an integer value \*/ 22 MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval3, (void \*) 17); 23 $^{24}$ B. Reading the attribute value in C 2526int flag, \*get\_val; 27struct foo \*get\_struct; 2829/\* Upon successful return, get\_val == &set\_val 30 (and therefore \*get\_val == 3) \*/  $^{31}$ MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval1, &get\_val, &flag); 32 /\* Upon successful return, get\_struct == &set\_struct \*/ 33 MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval2, &get\_struct, &flag); 34 /\* Upon successful return, get\_val == (void\*) 17 \*/ 35 i.e., (MPI\_Aint) get\_val == 17 \*/ /\* 36 MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval3, &get\_val, &flag); 37 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 38 39 LOGICAL FLAG 40INTEGER IERR, GET\_VAL, GET\_STRUCT 41 42! Upon successful return, GET\_VAL == &set\_val, possibly truncated 43 CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL1, GET\_VAL, FLAG, IERR) 44 ! Upon successful return, GET\_STRUCT == &set\_struct, possibly truncated 45CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL2, GET\_STRUCT, FLAG, IERR) 46 ! Upon successful return, GET\_VAL == 17 47CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL3, GET\_VAL, FLAG, IERR) 48

```
1
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                       \mathbf{2}
LOGICAL FLAG
                                                                                       3
INTEGER IERR
                                                                                       4
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
                                                                                       5
                                                                                       6
! Upon successful return, GET_VAL == &set_val
                                                                                       7
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
                                                                                       8
! Upon successful return, GET_STRUCT == &set_struct
                                                                                       9
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
                                                                                       10
! Upon successful return, GET_VAL == 17
                                                                                       11
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                       12
                                                                                       13
                                                                                       14
                                                                                       15
Example 17.14
                                                                                       16
    A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
                                                                                       17
                                                                                       18
INTEGER IERR, VAL
                                                                                       19
VAL = 7
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
                                                                                       20
                                                                                       21
    B. Reading the attribute value in C
                                                                                       22
                                                                                       23
int flag;
                                                                                       ^{24}
int *value;
                                                                                       25
                                                                                       26
/* Upon successful return, value points to internal MPI storage and
                                                                                       27
   *value == (int) 7 */
                                                                                       28
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
                                                                                       29
                                                                                       30
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                       31
                                                                                       32
LOGICAL FLAG
                                                                                       33
INTEGER IERR, VALUE
                                                                                       34
                                                                                       35
! Upon successful return, VALUE == 7
                                                                                       36
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                       37
                                                                                       38
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                       39
                                                                                       40
LOGICAL FLAG
                                                                                       41
INTEGER IERR
                                                                                       42
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
                                                                                       43
                                                                                       44
! Upon successful return, VALUE == 7 (sign extended)
                                                                                       45
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                       46
                                                                                       47
                                                                                       48
Example 17.15 A. Setting an attribute value via a Fortran MPI-2 call
```

```
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
\overline{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
         B. Reading the attribute value in C
10
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 == 2^40, 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
47
     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];

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

1

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 must be in Fortran the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI\_BOTTOM = 0 (Caveat: Defining MPI\_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI\_BOTTOM; it may be that MPI\_BOTTOM = 1 is better ...) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

14 15 16

# 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 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
44
        CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
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.

**Unofficial Draft for Comment Only** 

 $\mathbf{2}$ 

# 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.

Error classes	
C type: const int (or unnamed enum)	)
Fortran type: INTEGER	
MPI_SUCCESS	
MPI_ERR_BUFFER	
MPI_ERR_COUNT	
MPI_ERR_TYPE	
MPI_ERR_TAG	
MPI_ERR_COMM	
MPI_ERR_RANK	
MPI_ERR_REQUEST	
MPI_ERR_ROOT	
MPI_ERR_GROUP	
MPI_ERR_OP	
MPI_ERR_TOPOLOGY	
MPI_ERR_DIMS	
MPI_ERR_ARG	
MPI_ERR_UNKNOWN	
MPI_ERR_TRUNCATE	
MPI_ERR_OTHER	
MPI_ERR_INTERN	
MPI_ERR_PENDING	
(Continued on next page)	)

1	Error classes (continued)
2	C type: const int (or unnamed enum)
3	Fortran type: INTEGER
4	MPI_ERR_IN_STATUS
5	MPI_ERR_ACCESS
6	MPI_ERR_AMODE
7	MPI_ERR_ASSERT
8	MPI_ERR_BAD_FILE
9	
10	MPI_ERR_CONVERSION
11	MPI_ERR_DISP
12	MPI_ERR_DUP_DATAREP
13	MPI_ERR_FILE_EXISTS
14	MPI_ERR_FILE_IN_USE
15	MPI_ERR_FILE
16	MPI_ERR_INFO_KEY
17	MPI_ERR_INFO_KEY MPI_ERR_INFO_NOKEY
18	
19	MPI_ERR_INFO_VALUE
	MPI_ERR_INFO
20	MPI_ERR_IO
21	MPI_ERR_KEYVAL
22	MPI_ERR_LOCKTYPE
23	MPI_ERR_NAME
24	MPI_ERR_NO_MEM
25	MPI_ERR_NOT_SAME
26	MPI_ERR_NO_SPACE
27	MPI_ERR_NO_SUCH_FILE
28	MPI_ERR_PORT
29	MPI_ERR_QUOTA
30	MPI_ERR_READ_ONLY
31	MPI_ERR_RMA_ATTACH
32	MPI_ERR_RMA_CONFLICT
33	MPI_ERR_RMA_RANGE
34	MPI_ERR_RMA_SHARED
35	MPI_ERR_RMA_SYNC
36	MPI_ERR_RMA_WRONG_FLAVOR
37	MPI_ERR_SERVICE
38	MPI_ERR_SIZE
39	MPI_ERR_SPAWN
40	MPI_ERR_UNSUPPORTED_DATAREP
41	MPI_ERR_UNSUPPORTED_OPERATION
42	MPI_ERR_WIN
43	
	(Continued on next page)
44	
45	
46	
47	
48	

Error classes (continued)	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	4
	4 5
MPI_T_ERR_NOT_INITIALIZED	6
MPI_T_ERR_MEMORY MPI_T_ERR_INVALID_INDEX	7
MPI_T_ERR_INVALID_INDEX	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
	20
Buffer Address Constants	21
C type: void * const	22
Fortran type: (predefined memory location) <sup>1</sup>	23
MPI_BOTTOM	24
MPI_IN_PLACE	25
<sup>1</sup> Note that in Fortran these constants are not usable for initialization	26
expressions or assignment. See Section $2.5.4$ on page 15.	27
	28
	29
Assorted Constants	30
C type: const int (or unnamed enum)	31
Fortran type: INTEGER	32
MPI_PROC_NULL	33
MPI_ANY_SOURCE	34
MPI_ANY_TAG	35
MPI_UNDEFINED	36
MPI_BSEND_OVERHEAD	37
MPI_KEYVAL_INVALID	38
MPI_LOCK_EXCLUSIVE	39
MPI_LOCK_SHARED	40
MPI_ROOT	41
	42
	43
	44
No Process Message Handle	45
C type: MPI_Message	46 47
Fortran type: INTEGER or TYPE(MPI_Message)	47
MPI_MESSAGE_NO_PROC	0±

Fortron Support Mathad Spacific Constants
Fortran Support Method Specific Constants
Fortran type: LOGICAL
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
Status size and reserved index values (Fortran only)
Fortran type: INTEGER
MPI_STATUS_SIZE
MPI_SOURCE
MPI_TAG
MPI_ERROR
Variable Address Size (Fortran only)
Fortran type: INTEGER
MPI_ADDRESS_KIND
MPI_COUNT_KIND
MPI_INTEGER_KIND
MPI_OFFSET_KIND
Error-handling specifiers
C type: MPI_Errhandler
• -
Fortran type: INTEGER or TYPE(MPI_Errhandler) MPI_ERRORS_ARE_FATAL
MPI_ERRORS_RETURN
Maximum Sizes for Strings
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_MAX_DATAREP_STRING
MPI_MAX_ERROR_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_LIBRARY_VERSION_STRING
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_MAX_PROCESSOR_NAME

Named Predefined Datatypes C type: MPI_Datatype	C types	
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 that	
	(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	
MPI_UNSIGNED_LONG_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_OFFSET	MPI_Offset	
MPI_C_COMPLEX	float _Complex	
MPI_C_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)	

1	Named Predefined Datatypes	Fortr	ran	types
2	C type: MPI_Datatype			
3	Fortran type: INTEGER			
4	or TYPE(MPI_Datatype)			
5	MPI_INTEGER	INTEG	GER	
6	MPI_REAL	REAL		
7	MPI_DOUBLE_PRECISION	DOUBL	LE I	PRECISION
8	MPI_COMPLEX	COMPL	LEX	
9	MPI_LOGICAL	LOGIC	CAL	
10	MPI_CHARACTER	CHARA	ACTI	ER(1)
11	MPI_AINT	INTEGER (KIND=MPI_ADDRESS_KI		(KIND=MPI_ADDRESS_KIND)
12	MPI_COUNT	INTEGER (KIND=MPI_COUNT_KIND)		(KIND=MPI_COUNT_KIND)
13	MPI_OFFSET	INTEGER (KIND=MPI_OFFSET_KINI		(KIND=MPI_OFFSET_KIND)
14	MPI_BYTE	(any Fortran type)		tran type)
15	MPI_PACKED	(any ]	For	tran type)
16				
17				
18	Named Predefined Datatype	$\mathbf{es}^1 \mid \mathbf{C}$	C+-	$+  ext{ types}$
19	C type: MPI_Datatype			
20	Fortran type: INTEGER			
21	or TYPE(MPI_Datatype)			
22	MPI_CXX_BOOL bool		000]	-
23	MPI_CXX_FLOAT_COMPLEX	s	std:	:complex <float></float>
24	MPI_CXX_DOUBLE_COMPLEX std::complex <double></double>		:complex <double></double>	
25	MPI_CXX_LONG_DOUBLE_COMPI	_EX   s	std:	:complex <long double=""></long>
26	$^{-1}$ If an accompanying C++ comp	oiler is	mis	ssing, then the
27	MPI datatypes in this table are	e not de	efin	ed.
28				
29				
30	Optional datatypes (H	Fortrai	n)	Fortran types
31	$\mathrm{C} \ \mathrm{type}$ : MPI_Datatype			
32	Fortran type: INTEGER			
33	or TYPE(MPI_Datatype)			
34	MPI_DOUBLE_COMPLEX			DOUBLE COMPLEX
35	MPI_INTEGER1			INTEGER*1
36	MPI_INTEGER2			INTEGER*2
37	MPI_INTEGER4			INTEGER*4
38	MPI_INTEGER8		INTEGER*8	
39	MPI_INTEGER16		INTEGER*16	
40	MPI_REAL2			REAL*2
41	MPI_REAL4		REAL*4	
42	MPI_REAL8			REAL*8
43	MPI_REAL16			REAL*16
44	MPI_COMPLEX4			COMPLEX*4
45	MPI_COMPLEX8			COMPLEX*8
46	MPI_COMPLEX16			COMPLEX*16
47	MPI_COMPLEX32			COMPLEX*32
48				

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
	10
	11
Datatypes for reduction functions (Fortran)	12
C type: MPI_Datatype	13
Fortran type: INTEGER or TYPE(MPI_Datatype)	14
MPI_2REAL	15
MPI_2DOUBLE_PRECISION	16
MPI_2INTEGER	17
	18
	19
Reserved communicators	20
C type: MPI_Comm	21
Fortran type: INTEGER or TYPE(MPI_Comm)	22
MPI_COMM_WORLD	23
MPI_COMM_SELF	24
	25
	26
Communicator on lit toma constants	27
Communicator split type constants	28
C type: const int (or unnamed enum)	29
Fortran type: INTEGER	30
MPI_COMM_TYPE_SHARED	31
	32
Desults of communication and more	33 34
Results of communicator and group comparisons	34
C type: const int (or unnamed enum)	35
Fortran type: INTEGER	30
MPI_IDENT	37
MPI_CONGRUENT	
MPI_SIMILAR	39
MPI_UNEQUAL	40
	41
	42
Environmental inquiry info here	43
Environmental inquiry info key	44
C type: MPI_Info	45
Fortran type: INTEGER or TYPE(MPI_Info)	46
MPI_INFO_ENV	47

	670	ANNEX A. LANGUAGE BINDINGS SUM
1		Environmental inquiry keys
2		C type: const int (or unnamed enum)
3		Fortran type: INTEGER
4		MPI_TAG_UB
5		MPI_IO
6		MPI_HOST
7		MPI_WTIME_IS_GLOBAL
8		
9		
10		Collective Operations
11		C type: MPI_Op
12		Fortran type: INTEGER or TYPE(MPI_Op)
13		MPI_MAX
14		MPI_MIN
15		MPI_SUM
16		MPI_PROD
17		MPI_MAXLOC
18		MPI_MINLOC
19		MPI_BAND
20		MPI_BOR
21		MPI_BXOR
22 23		MPI_LAND
23 24		MPI_LOR
25		MPI_LXOR
26		MPI_REPLACE
20		MPI_NO_OP
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

Null Handles	1
	2
C/Fortran name	
C type / Fortran type	3
MPI_GROUP_NULL	4
<pre>MPI_Group / INTEGER or TYPE(MPI_Group)</pre>	5
MPI_COMM_NULL	6
MPI_Comm / INTEGER or TYPE(MPI_Comm)	7
MPI_DATATYPE_NULL	8
MPI_Datatype / INTEGER or TYPE(MPI_Datatype)	9
MPI_REQUEST_NULL	10
MPI_Request / INTEGER or TYPE(MPI_Request)	11
MPI_OP_NULL	12
MPI_Op / INTEGER or TYPE(MPI_Op)	13
MPI_ERRHANDLER_NULL	14
MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)	15
MPI_FILE_NULL	16
<pre>MPI_File / INTEGER or TYPE(MPI_File)</pre>	17
MPI_INFO_NULL	18
<pre>MPI_Info / INTEGER or TYPE(MPI_Info)</pre>	19
MPI_WIN_NULL	20
MPI_Win / INTEGER or TYPE(MPI_Win)	21
MPI_MESSAGE_NULL	22
MPI_Message / INTEGER or TYPE(MPI_Message)	23
<b>_</b>	24

C type: MPI_G	roup
Fortran type: 1	INTEGER or TYPE(MPI_Group)
MPI_GROUP_I	ЕМРТҮ

Topologies
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_GRAPH
MPI_CART
MPI_DIST_GRAPH

 $^{29}$ 

1	Predefined functions
2 3	C/Fortran name
4	C type
5	/ Fortran type with mpi module / Fortran type with mpi_f08 module
6	MPI_COMM_NULL_COPY_FN
7	MPI_Comm_copy_attr_function
8	/ COMM_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Comm_copy_attr_function) <sup>1</sup> )
9	MPI_COMM_DUP_FN
10	MPI_Comm_copy_attr_function
11	$/ \text{COMM\_COPY\_ATTR\_FUNCTION} / \text{PROCEDURE(MPI\_Comm\_copy\_attr\_function)}^{1}$
12	MPI_COMM_NULL_DELETE_FN
13	MPI_Comm_delete_attr_function
14	$/$ COMM_DELETE_ATTR_FUNCTION $/$ PROCEDURE(MPI_Comm_delete_attr_function) $^1$ )
15	MPI_WIN_NULL_COPY_FN
16	MPI_Win_copy_attr_function
17	/ WIN_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Win_copy_attr_function) <sup>1</sup> )
18	MPI_WIN_DUP_FN
19	MPI_Win_copy_attr_function
20	/ WIN_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Win_copy_attr_function) <sup>1</sup> )
21 22	MPI_WIN_NULL_DELETE_FN MPI_Win_delete_attr_function
22	/ WIN_DELETE_ATTR_FUNCTION / PROCEDURE(MPI_Win_delete_attr_function) <sup>1</sup> )
24	MPI_TYPE_NULL_COPY_FN
25	MPI_Type_copy_attr_function
26	/ TYPE_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Type_copy_attr_function) <sup>1</sup> )
27	MPI_TYPE_DUP_FN
28	MPI_Type_copy_attr_function
29	$/$ TYPE_COPY_ATTR_FUNCTION $/$ PROCEDURE(MPI_Type_copy_attr_function) $^{1}$ )
30	MPI_TYPE_NULL_DELETE_FN
31	MPI_Type_delete_attr_function
32	/ TYPE_DELETE_ATTR_FUNCTION / PROCEDURE(MPI_Type_delete_attr_function) <sup>1</sup> )
33	MPI_CONVERSION_FN_NULL
34	MPI_Datarep_conversion_function
35 36	/ DATAREP_CONVERSION_FUNCTION / PROCEDURE(MPI_Datarep_conversion_function) <sup>1</sup> )
37	<sup>-1</sup> See the advice to implementors (on page 270) and advice to users (on page 270) on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, in
38	Section 6.7.2 on page 267.
39	Section 0.1.2 on page 201.
40	
41	
42	
43	
44	
45	
46	
47	
48	

Deprecated predefined functions	
C/Fortran name	
C type / Fortran type with $\mathtt{mpi}$ module	
MPI_NULL_COPY_FN	
MPI_Copy_function / COPY_FUNCTION	
MPI_DUP_FN	
MPI_Copy_function / COPY_FUNCTION	
MPI_NULL_DELETE_FN	
MPI_Delete_function / DELETE_FUNCTION	JN
Predefined Attribute Keys	
C type: const int (or unnamed enum)	
Fortran type: INTEGER	
MPI_APPNUM	
MPI_LASTUSEDCODE	
MPI_UNIVERSE_SIZE	
MPI WIN BASE	

MPI_UNIVERSE_SIZE
MPI_WIN_BASE
MPI_WIN_DISP_UNIT
MPI_WIN_SIZE
MPI_WIN_CREATE_FLAVOR
MPI_WIN_MODEL

## **MPI** Window Create Flavors C type: const int (or unnamed enum) Fortran type: INTEGER MPI\_WIN\_FLAVOR\_CREATE MPI\_WIN\_FLAVOR\_ALLOCATE MPI\_WIN\_FLAVOR\_DYNAMIC MPI\_WIN\_FLAVOR\_SHARED

### **MPI** Window Models

C type: const int (or unnamed enum) Fortran type: INTEGER MPI\_WIN\_SEPARATE MPI\_WIN\_UNIFIED

1		Mode Constants
2		C type: const int (or unnamed enum)
3		Fortran type: INTEGER
4		MPI_MODE_APPEND
5		MPI_MODE_CREATE
6		MPI_MODE_DELETE_ON_CLOSE
7		MPI_MODE_EXCL
8		
9		MPI_MODE_NOCHECK
10		MPI_MODE_NOPRECEDE
1		
1:		MPI_MODE_NOSUCCEED
1:		MPI_MODE_RDONLY
14		MPI_MODE_RDWR
1		MPI_MODE_SEQUENTIAL
10		MPI_MODE_UNIQUE_OPEN
1'		MPI_MODE_WRONLY
18		
19		
20		Datatype Decoding Constants
2		C type: const int (or unnamed enum)
23		Fortran type: INTEGER
23	1	MPI_COMBINER_CONTIGUOUS
2	L.	MPI_COMBINER_DARRAY
2	;	MPI_COMBINER_DUP
20	;	MPI_COMBINER_F90_COMPLEX
2'		MPI_COMBINER_F90_INTEGER
28		MPI_COMBINER_F90_REAL
29	,	MPI_COMBINER_HINDEXED
30	1	MPI_COMBINER_HVECTOR
3		MPI_COMBINER_INDEXED_BLOCK
35		MPI_COMBINER_HINDEXED_BLOCK
33		MPI_COMBINER_INDEXED
34	L	MPI_COMBINER_NAMED
3		MPI_COMBINER_RESIZED
30		MPI_COMBINER_STRUCT
3'		MPI_COMBINER_SUBARRAY
38		
		MPI_COMBINER_VECTOR
39		
40		Three de Constants
4		Threads Constants
4:	2	C type: const int (or unnamed enum)
43		Fortran type: INTEGER
44	L	MPI_THREAD_FUNNELED
4		MPI_THREAD_MULTIPLE
40	5	MPI_THREAD_SERIALIZED
4'		MPI_THREAD_SINGLE
48	3	

File Operation Constants, Part 1	1
C type: const MPI_Offset (or unnamed enum)	2
Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)	3
MPI_DISPLACEMENT_CURRENT	4
	5
	6
File Operation Constants, Part 2	7
C type: const int (or unnamed enum)	8
Fortran type: INTEGER	9
MPI_DISTRIBUTE_BLOCK	10
MPI_DISTRIBUTE_CYCLIC	11
MPI_DISTRIBUTE_DFLT_DARG	12
MPI_DISTRIBUTE_NONE	
MPI_ORDER_C	14
MPI_ORDER_FORTRAN	14
MPI_SEEK_CUR	16
MPI_SEEK_END	
MPI_SEEK_SET	17
	18
	19
F90 Datatype Matching Constants	20
C type: const int (or unnamed enum)	21
Fortran type: INTEGER	22
MPI_TYPECLASS_COMPLEX	23
MPI_TYPECLASS_INTEGER	24
MPI_TYPECLASS_REAL	25
	26
	27
Constants Specifying Empty or Ignored Input	28
C/Fortran name	29
C type / Fortran type <sup>1</sup>	30
MPI_ARGVS_NULL	31
<pre>char*** / 2-dim. array of CHARACTER*(*)</pre>	32
MPI_ARGV_NULL	33
<pre>char** / array of CHARACTER*(*)</pre>	34
MPI_ERRCODES_IGNORE	35
int* / INTEGER array	36
MPI_STATUSES_IGNORE	37
<pre>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</pre>	38
or TYPE(MPI_Status), DIMENSION(*)	39
MPI_STATUS_IGNORE	40
MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)	41
or TYPE(MPI_Status)	42
MPI_UNWEIGHTED	43
int* / INTEGER array	44
MPI_WEIGHTS_EMPTY	45
int* / INTEGER array	46
<sup>1</sup> Note that in Fortran these constants are not usable for initialization	47
expressions or assignment. See Section 2.5.4 on page 15	48

expressions or assignment. See Section 2.5.4 on page 15.

1	C Constants Specifyir	ng Ignored Input (no Fortran)
2		equivalent to Fortran
3		MPI_STATUSES_IGNORE in mpi / mpif.h
4	MPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h
5	C type: MPI_F08_status*	equivalent to Fortran
6	MPI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08
7		MPI_STATUS_IGNORE in mpi_f08
8		<u> </u>
9		
10	C preprocessor Const	ants and Fortran Parameters
11	C type: const int (or un	
12	Fortran type: INTEGER	,
13	MPI_SUBVERSION	
14	MPI_VERSION	
15		
16		
17	Null handles used in the	e MPI tool information interface
18	MPI_T_ENUM_NULL	
19	MPI_T_enum	
20	MPI_T_CVAR_HANDLE_NUL	L
21	MPI_T_cvar_handle	
22	MPI_T_PVAR_HANDLE_NUL	L
23	MPI_T_pvar_handle	_
24	MPI_T_PVAR_SESSION_NUL	L
25	MPI_T_pvar_session	_
26	r	
27		
28	Verbosity Levels in the	MPI tool information interface
29	C type: const int (or unnar	
30	Fortran type: INTEGER	
31	MPI_T_VERBOSITY_USER_I	BASIC
32	MPI_T_VERBOSITY_USER_I	
33	MPI_T_VERBOSITY_USER_/	
34	MPI_T_VERBOSITY_TUNER	
35	MPI_T_VERBOSITY_TUNER	_
36	MPI_T_VERBOSITY_TUNER	_
37	MPI_T_VERBOSITY_MPIDE	_
38	MPI_T_VERBOSITY_MPIDE	—
39	MPI_T_VERBOSITY_MPIDE	—
40		
41		
42		
43		
44		
45		
46		
47		
48		

	_
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	
the MPI tool information interface type: const int (or unnamed enum)	
ortran type: INTEGER	
IPI_T_SCOPE_CONSTANT	
IPI_T_SCOPE_READONLY	
PI_T_SCOPE_LOCAL	
IPI_T_SCOPE_GROUP	
1PI_T_SCOPE_GROUP_EQ 1PI_T_SCOPE_ALL	
IPI_T_SCOPE_ALL_EQ	
Additional constants used by the MPI tool information interface	

1	Performance variables classes used by the
2	MPI tool information interface
3	C type: const int (or unnamed enum)
4	Fortran type: INTEGER
5	MPI_T_PVAR_CLASS_STATE
6	MPI_T_PVAR_CLASS_LEVEL
7	MPI_T_PVAR_CLASS_SIZE
8	MPI_T_PVAR_CLASS_PERCENTAGE
9	MPI_T_PVAR_CLASS_HIGHWATERMARK
10	MPI_T_PVAR_CLASS_LOWWATERMARK
11	MPI_T_PVAR_CLASS_COUNTER
12	MPI_T_PVAR_CLASS_AGGREGATE
13	MPI_T_PVAR_CLASS_AGGREGATE
14	MPI_T_PVAR_CLASS_TIMER MPI_T_PVAR_CLASS_GENERIC
15	
16	
17	A.1.2 Types
18	A.I.2 Types
19	The following are defined C type definitions, included in the file mpi.h.
20	/* C opaque types */
21	MPI_Aint
22	MPI_Count
23	MPI_Fint
24	MPI_Offset
25	
26	MPI_Status
20 27	MPI_F08_status
27	(* C handles to asserted structures */
28 29	/* C handles to assorted structures */
30	MPI_Comm
31	MPI_Datatype
32	MPI_Errhandler
33	MPI_File
34	MPI_Group
	MPI_Info
35 36	MPI_Message
37	MPI_Op
	MPI_Request
38	MPI_Win
39 40	
40	<pre>/* Types for the MPI_T interface */</pre>
41	MPI_T_enum
42	MPI_T_cvar_handle
43	MPI_T_pvar_handle
44	MPI_T_pvar_session
45	
46	
47	The following are defined Fortran type definitions, included in the mpi_f08 and mpi
48	module.

1 ! Fortran opaque types in the mpi\_f08 and mpi module 2 TYPE(MPI\_Status) 3 ! Fortran handles in the mpi\_f08 and mpi module 4 TYPE(MPI\_Comm) 5 6 TYPE(MPI\_Datatype) TYPE(MPI\_Errhandler) 7 TYPE(MPI\_File) 9 TYPE(MPI\_Group) 10 TYPE(MPI\_Info) 11 TYPE(MPI\_Op) TYPE(MPI\_Request) 1213 TYPE(MPI\_Win) 1415A.1.3 Prototype Definitions 16C Bindings 1718 The following are defined C typedefs for user-defined functions, also included in the file 19 mpi.h. 2021/\* prototypes for user-defined functions \*/ typedef void MPI\_User\_function(void \*invec, void \*inoutvec, int \*len, 22 23MPI\_Datatype \*datatype); 2425typedef int MPI\_Comm\_copy\_attr\_function(MPI\_Comm oldcomm, 26int comm\_keyval, void \*extra\_state, void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag); 2728 typedef int MPI\_Comm\_delete\_attr\_function(MPI\_Comm comm, 29int comm\_keyval, void \*attribute\_val, void \*extra\_state); 30 31typedef int MPI\_Win\_copy\_attr\_function(MPI\_Win oldwin, int win\_keyval, 32 void \*extra\_state, void \*attribute\_val\_in, 33 void \*attribute\_val\_out, int \*flag); 34 typedef int MPI\_Win\_delete\_attr\_function(MPI\_Win win, int win\_keyval, 35 void \*attribute\_val, void \*extra\_state); 36 37 typedef int MPI\_Type\_copy\_attr\_function(MPI\_Datatype oldtype, 38 int type\_keyval, void \*extra\_state, 39 void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag); 40typedef int MPI\_Type\_delete\_attr\_function(MPI\_Datatype datatype, 41 int type\_keyval, void \*attribute\_val, void \*extra\_state); 42typedef void MPI\_Comm\_errhandler\_function(MPI\_Comm \*, int \*, ...); 43 44typedef void MPI\_Win\_errhandler\_function(MPI\_Win \*, int \*, ...); typedef void MPI\_File\_errhandler\_function(MPI\_File \*, int \*, ...); 454647typedef int MPI\_Grequest\_query\_function(void \*extra\_state, 48 MPI\_Status \*status);

Unofficial Draft for Comment Only

```
1
     typedef int MPI_Grequest_free_function(void *extra_state);
\mathbf{2}
     typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
3
4
     typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
5
                  MPI_Aint *file_extent, void *extra_state);
6
     typedef int MPI_Datarep_conversion_function(void *userbuf,
7
                 MPI_Datatype datatype, int count, void *filebuf,
8
                  MPI_Offset position, void *extra_state);
9
10
11
     Fortran 2008 Bindings with the mpi_f08 Module
12
         The callback prototypes when using the Fortran mpi_f08 module are shown below:
13
14
         The user-function argument to MPI_Op_create should be declared according to:
15
     ABSTRACT INTERFACE
16
       SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)
17
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
18
           TYPE(C_PTR), VALUE :: invec, inoutvec
19
           INTEGER :: len
20
           TYPE(MPI_Datatype) :: datatype
21
         The copy and delete function arguments to MPI_Comm_create_keyval should be de-
22
     clared according to:
23
     ABSTRACT INTERFACE
24
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
25
       attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
26
           TYPE(MPI_Comm) :: oldcomm
27
           INTEGER :: comm_keyval, ierror
28
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
29
           attribute_val_out
30
           LOGICAL :: flag
31
32
     ABSTRACT INTERFACE
33
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
34
       attribute_val, extra_state, ierror) BIND(C)
35
           TYPE(MPI_Comm) :: comm
36
           INTEGER :: comm_keyval, ierror
37
           INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
38
         The copy and delete function arguments to MPI_Win_create_keyval should be declared
39
     according to:
40
     ABSTRACT INTERFACE
41
       SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
42
       attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
43
           TYPE(MPI_Win) :: oldwin
44
           INTEGER :: win_keyval, ierror
45
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
46
           attribute_val_out
47
           LOGICAL :: flag
48
```

```
1
ABSTRACT INTERFACE
                                                                                      \mathbf{2}
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
  extra_state, ierror) BIND(C)
      TYPE(MPI_Win) :: win
                                                                                      4
      INTEGER :: win_keyval, ierror
                                                                                      5
                                                                                      6
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                      7
    The copy and delete function arguments to MPI_Type_create_keyval should be declared
                                                                                      8
according to:
                                                                                      9
ABSTRACT INTERFACE
                                                                                      10
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                      11
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
                                                                                      12
      TYPE(MPI_Datatype) :: oldtype
                                                                                      13
      INTEGER :: type_keyval, ierror
                                                                                      14
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                      15
      attribute_val_out
                                                                                      16
      LOGICAL :: flag
                                                                                      17
                                                                                      18
ABSTRACT INTERFACE
                                                                                      19
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                      20
  attribute_val, extra_state, ierror) BIND(C)
                                                                                      21
      TYPE(MPI_Datatype) :: datatype
                                                                                      22
      INTEGER :: type_keyval, ierror
                                                                                      23
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                      ^{24}
    The handler-function argument to MPI_Comm_create_errhandler should be declared
                                                                                      25
like this:
                                                                                      26
ABSTRACT INTERFACE
                                                                                      27
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)
                                                                                      28
      TYPE(MPI_Comm) :: comm
                                                                                      29
      INTEGER :: error_code
                                                                                      30
                                                                                      31
    The handler-function argument to MPI_Win_create_errhandler should be declared like
                                                                                      32
this:
                                                                                      33
ABSTRACT INTERFACE
                                                                                      34
  SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)
                                                                                      35
      TYPE(MPI_Win) :: win
                                                                                      36
      INTEGER :: error_code
                                                                                      37
    The handler-function argument to MPI_File_create_errhandler should be declared like
                                                                                      38
this:
                                                                                      39
ABSTRACT INTERFACE
                                                                                      40
  SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)
                                                                                      41
      TYPE(MPI_File) :: file
                                                                                      42
      INTEGER :: error_code
                                                                                      43
                                                                                      44
    The query, free, and cancel function arguments to MPI_Grequest_start should be de-
                                                                                      45
clared according to:
                                                                                      46
ABSTRACT INTERFACE
                                                                                      47
                                                                                      48
```

```
1
       SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
\mathbf{2}
       BIND(C)
3
           TYPE(MPI_Status) :: status
4
           INTEGER :: ierror
5
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
6
     ABSTRACT INTERFACE
7
       SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)
8
           INTEGER :: ierror
9
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
10
11
     ABSTRACT INTERFACE
12
       SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
13
       BIND(C)
14
           INTEGER :: ierror
15
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
16
           LOGICAL :: complete
17
         The extend and conversion function arguments to MPI_Register_datarep should be de-
18
     clared according to:
19
     ABSTRACT INTERFACE
20
       SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
21
       ierror) BIND(C)
22
           TYPE(MPI_Datatype) :: datatype
23
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
24
           INTEGER :: ierror
25
26
     ABSTRACT INTERFACE
27
       SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
28
       filebuf, position, extra_state, ierror) BIND(C)
29
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
30
           TYPE(C_PTR), VALUE :: userbuf, filebuf
31
           TYPE(MPI_Datatype) :: datatype
32
           INTEGER :: count, ierror
33
           INTEGER(KIND=MPI_OFFSET_KIND) :: position
34
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
35
     Fortran Bindings with mpif.h or the mpi Module
36
37
         With the Fortran mpi module or mpif.h, here are examples of how each of the user-
38
     defined subroutines should be declared.
39
         The user-function argument to MPI_OP_CREATE should be declared like this:
40
     SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
41
        <type> INVEC(LEN), INOUTVEC(LEN)
42
        INTEGER LEN, DATATYPE
43
44
         The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
45
     declared like these:
46
47
     SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
48
                   ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
```

```
1
   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
                                                                                    \mathbf{2}
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT
   LOGICAL FLAG
                                                                                    5
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                                                                                    6
             EXTRA_STATE, IERROR)
   INTEGER COMM, COMM_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                    9
                                                                                    10
    The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be
                                                                                    11
declared like these:
                                                                                    12
                                                                                    13
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                                                                                    14
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                    15
   INTEGER OLDWIN, WIN_KEYVAL, IERROR
                                                                                    16
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                    17
             ATTRIBUTE_VAL_OUT
                                                                                    18
   LOGICAL FLAG
                                                                                    19
                                                                                    20
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
                                                                                    21
             EXTRA_STATE, IERROR)
                                                                                    22
   INTEGER WIN, WIN_KEYVAL, IERROR
                                                                                    23
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                    24
                                                                                    25
    The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be
                                                                                    26
declared like these:
                                                                                    27
                                                                                    28
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                    29
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                    30
   INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
                                                                                    31
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
                                                                                    32
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
                                                                                    33
   LOGICAL FLAG
                                                                                    34
                                                                                    35
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                    36
              EXTRA_STATE, IERROR)
                                                                                    37
   INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                    38
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                    39
                                                                                    40
    The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-
                                                                                    41
clared like this:
                                                                                    42
                                                                                    43
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
                                                                                    44
   INTEGER COMM, ERROR_CODE
                                                                                    45
                                                                                    46
   The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-
                                                                                    47
clared like this:
                                                                                    48
```

1 2 3	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE
4 5	The handler-function argument to $MPI\_FILE\_CREATE\_ERRHANDLER$ should be declared like this:
6 7 8	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE
9 10 11	The query, free, and cancel function arguments to $MPI\_GREQUEST\_START$ should be declared like these:
12 13 14 15	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
16 17	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR) INTEGER IERROR
18 19	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
20 21 22 23	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) INTEGER IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
24 25 26	LOGICAL COMPLETE The extend and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:
27 28 29 30	SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
31 32 33 34 35 36 37	SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, EXTRA_STATE, IERROR) <type> USERBUF(*), FILEBUF(*) INTEGER COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) POSITION INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</type>
38 39	A.1.4 Deprecated Prototype Definitions
40 41 42	The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.
43 44 45	<pre>/* prototypes for user-defined functions */ typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,</pre>
46 47 48	<pre>void *attribute_val_out, int *flag); typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state);</pre>

The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI\_KEYVAL\_CREATE should be declared like these:

	4
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,	5
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)	6
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	7
ATTRIBUTE_VAL_OUT, IERR	8
LOGICAL FLAG	9
	10
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)	11
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	12
	13
A.1.5 Info Keys	14
access_style	15
appnum	16
arch	17
cb_block_size	18
cb_buffer_size	19
cb_nodes	20
chunked_item	21
chunked_size	22
chunked	23
collective_buffering	24
file_perm	25
filename	26
file	27
host	28
io_node_list	29
ip_address	30
ip_port	31 32
nb_proc	32
no_locks	34
num_io_nodes	34
path	36
soft	37
striping_factor	38
striping_unit	39
wdir	40
	41
	42
A.1.6 Info Values	43
false	44
random	45
read_mostly	46
read_once	47
reverse_sequential	48

 $\mathbf{2}$ 

1	sequential
2	true
3	write_mostly
4	write_once
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
22	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

A.2 C Bindings	1
A.2.1 Point-to-Point Communication C Bindings	2 3
int MPI_Bsend_init(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	4 5 6
int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	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
int MPI_Cancel(MPI_Request *request)	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
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
<pre>int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,</pre>	26 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>	31 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_init(void* buf, int count, MPI_Datatype datatype, int source,</pre>	43 44 45
<pre>int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,</pre>	46 47 48

```
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_init(const void* buf, int count, MPI_Datatype datatype,
6
                   int dest, int tag, MPI_Comm comm, MPI_Request *request)
7
     int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest,
8
                   int tag, MPI_Comm comm)
9
10
     int MPI_Send_init(const void* buf, int count, MPI_Datatype datatype,
11
                   int dest, int tag, MPI_Comm comm, MPI_Request *request)
12
     int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
13
                   int tag, MPI_Comm comm)
14
15
     int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
16
                   int dest, int sendtag, void *recvbuf, int recvcount,
17
                   MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
18
                   MPI_Status *status)
19
     int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
20
                   int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
21
                   MPI_Status *status)
22
23
     int MPI_Ssend_init(const void* buf, int count, MPI_Datatype datatype,
^{24}
                   int dest, int tag, MPI_Comm comm, MPI_Request *request)
25
     int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest,
26
                   int tag, MPI_Comm comm)
27
28
     int MPI_Startall(int count, MPI_Request array_of_requests[])
29
     int MPI_Start(MPI_Request *request)
30
31
     int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
32
                   MPI_Status array_of_statuses[])
33
34
     int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,
                   int *flag, MPI_Status *status)
35
36
     int MPI_Test_cancelled(const MPI_Status *status, int *flag)
37
38
     int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
39
     int MPI_Testsome(int incount, MPI_Request array_of_requests[],
40
                   int *outcount, int array_of_indices[],
41
                   MPI_Status array_of_statuses[])
42
43
     int MPI_Waitall(int count, MPI_Request array_of_requests[],
44
                   MPI_Status array_of_statuses[])
45
     int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,
46
                   MPI_Status *status)
47
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

688

int	MPI_Wait(MPI_Request *request, MPI_Status *status)	1
int	<pre>MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>	2 3
	<pre>int *outcount, int array_of_indices[],</pre>	4
	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>	13 14
int	<pre>MPI_Pack_external(const char datarep[], const void *inbuf, int incount,</pre>	15
1110	MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,	16
	MPI_Aint *position)	17 18
int	MPI_Pack_external_size(const char datarep[], int incount,	19
	MPI_Datatype datatype, MPI_Aint *size)	20
		21
int	MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype,	22
	<pre>void *outbuf, int outsize, int *position, MPI_Comm comm)</pre>	23
int	MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,	24
	int *size)	25
int	MPI_Type_commit(MPI_Datatype *datatype)	26
		27
int	MPI_Type_contiguous(int count, MPI_Datatype oldtype,	28
	MPI_Datatype *newtype)	29
int	MPI_Type_create_darray(int size, int rank, int ndims, const	30
	int array_of_gsizes[], const int array_of_distribs[], const	31 32
	<pre>int array_of_dargs[], const int array_of_psizes[], int order,</pre>	32
	MPI_Datatype oldtype, MPI_Datatype *newtype)	34
		35
TUC	MPI_Type_create_hindexed_block(int count, int blocklength, const	36
	<pre>MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	37
	MFI_Datatype *newtype)	38
int	<pre>MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],</pre>	39
	<pre>const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,</pre>	40
	MPI_Datatype *newtype)	41
int	MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,	42
	MPI_Datatype oldtype, MPI_Datatype *newtype)	43
		44
int	MPI_Type_create_indexed_block(int count, int blocklength, const	45
	<pre>int array_of_displacements[], MPI_Datatype oldtype, NDI D to the second se</pre>	46
	MPI_Datatype *newtype)	47
		48

1 2	int	<pre>MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint</pre>
3 4 5 6	int	<pre>MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>
7 8 9 10	int	<pre>MPI_Type_create_subarray(int ndims, const int array_of_sizes[], const int array_of_subsizes[], const int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>
11	int	MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
12	int	MPI_Type_free(MPI_Datatype *datatype)
13 14 15 16 17	int	<pre>MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,</pre>
18 19 20	int	<pre>MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>
21 22	int	<pre>MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)</pre>
23 24 25	int	<pre>MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb, MPI_Count *extent)</pre>
26 27	int	<pre>MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)</pre>
28 29 30	int	<pre>MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb, MPI_Count *true_extent)</pre>
31 32 33	int	<pre>MPI_Type_indexed(int count, const int array_of_blocklengths[], const</pre>
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 42	int	<pre>MPI_Unpack_external(const char datarep[], const void *inbuf, MPI_Aint insize, MPI_Aint *position, void *outbuf, int outcount, MPI_Datatype datatype)</pre>
43 44 45 46 47	int	<pre>MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf,</pre>
47 48		

A.2.	.3 Collecti	ve Communication C Bindings	1
int	MPI_Allga	ther(const void* sendbuf, int sendcount,	2 3
		MPI_Datatype sendtype, void* recvbuf, int recvcount,	4
		MPI_Datatype recvtype, MPI_Comm comm)	5
int	MPI_Allga	therv(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	MP1_Allre	duce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	10 11
int	MPI_Allto	<pre>all(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>	12 13 14
int	MPI_Allto	<pre>pallv(const void* sendbuf, const int sendcounts[], const</pre>	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
TUC	MPI_AIICO	<pre>vallw(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_Barri	er(MPI_Comm comm)	25
int	MPI_Bcast	(void* buffer, int count, MPI_Datatype datatype, int root,	26
		MPI_Comm comm)	27 28
int	MPI_Exsca	n(const void* sendbuf, void* recvbuf, int count,	29
		MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	30
int	MPI_Gathe	er(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	31 32
		<pre>void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,</pre>	33
		MPI_Comm comm)	34
int	MPI_Gathe	erv(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
	MDT Tolla		38
TUC	MP1_1allg	ather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount,	39
		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	40 41
int	MPI_Iallg	atherv(const void* sendbuf, int sendcount,	41
	0	<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],</pre>	43
		<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	44
		MPI_Request* request)	45 46
int	MPI_Iallr	reduce(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	<pre>int MPI_Ialltoall(const void* sendbuf, int sendcount,</pre>
6 7 8 9 10	<pre>int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], const</pre>
11 12 13 14	<pre>int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], const</pre>
15 16	<pre>int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)</pre>
17 18	<pre>int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,</pre>
19 20 21 22	<pre>int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,</pre>
23 24 25	<pre>int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
26 27 28 29 30	<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>
31 32 33	<pre>int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count,</pre>
34 35 36 37	<pre>int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>
38 39 40	<pre>int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, const</pre>
41 42 43 44	<pre>int MPI_Iscan(const void* sendbuf, void* recvbuf, int count,</pre>
45 46 47 48	<pre>int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>

		1
int	<pre>MPI_Iscatterv(const void* sendbuf, const int sendcounts[], const</pre>	1
	<pre>int displs[], MPI_Datatype sendtype, void* recvbuf,</pre>	2
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,	3
	MPI_Request *request)	4
int	MPI_Op_commutative(MPI_Op op, int *commute)	5
		6
int	<pre>MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)</pre>	7 8
int	MPI_Op_free(MPI_Op *op)	9
		10
int	MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,	11
	MPI_Datatype datatype, MPI_Op op)	12
int	MPI_Reduce(const void* sendbuf, void* recvbuf, int count,	13
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	14
		15
int	<pre>MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>	16
	<pre>int recvcount, MPI_Datatype datatype, MPI_Op op, </pre>	17
	MPI_Comm comm)	18
int	MPI_Reduce_scatter(const void* sendbuf, void* recvbuf, const	19
	<pre>int recvcounts[], MPI_Datatype datatype, MPI_Op op,</pre>	20
	MPI_Comm comm)	21
		22
int	MPI_Scan(const void* sendbuf, void* recvbuf, int count,	23
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	24
int	MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	25
	<pre>void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,</pre>	26
	MPI_Comm comm)	27
		28
int	MPI_Scatterv(const void* sendbuf, const int sendcounts[], const	29
	<pre>int displs[], MPI_Datatype sendtype, void* recvbuf,</pre>	30
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	31
		32
A.2.	4 Groups, Contexts, Communicators, and Caching C Bindings	33
		34
int	<pre>MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)</pre>	35
int	MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,	36
	MPI_Comm *newcomm)	37
		38
int	MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,	39
	MPI_Comm_delete_attr_function *comm_delete_attr_fn,	40
	<pre>int *comm_keyval, void *extra_state)</pre>	41
int	MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)	42
in+	MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)	43
1110	In I_comm_derete_attr (In I_comm comm, Int comm_Keyvar)	44
int	MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,	45
	<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	46
int	MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)	47
TTT 0		48

```
1
     int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
\mathbf{2}
     int MPI_Comm_free_keyval(int *comm_keyval)
3
4
     int MPI_Comm_free(MPI_Comm *comm)
5
     int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
6
                   int *flag)
7
8
     int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
9
     int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
10
11
     int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
12
     int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
13
14
     int MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,
15
                   void *extra_state, void *attribute_val_in,
16
                   void *attribute_val_out, int *flag)
17
     int MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void
18
                   *attribute_val, void *extra_state)
19
20
     int MPI_Comm_rank(MPI_Comm comm, int *rank)
21
     int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
22
23
     int MPI_Comm_remote_size(MPI_Comm comm, int *size)
24
     int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
25
26
     int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
27
28
     int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
29
     int MPI_Comm_size(MPI_Comm comm, int *size)
30
^{31}
     int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
32
     int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
33
                   MPI_Info info, MPI_Comm *newcomm)
34
35
     int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
36
     int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
37
38
     int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
39
                   MPI_Group *newgroup)
40
     int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
41
                   MPI_Group *newgroup)
42
43
     int MPI_Group_free(MPI_Group *group)
44
     int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
45
                   MPI_Group *newgroup)
46
47
48
```

int	<pre>MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	1 2
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
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_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	23 24 25
int	MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)	26 27
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>	28 29 30
int	MPI_Type_free_keyval(int *type_keyval)	31 32
int	<pre>MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void      *attribute_val, int *flag)</pre>	33 34
int	<pre>MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int</pre>	35 36 37
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>	38 39 40
int	<pre>MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void     *attribute_val, void *extra_state)</pre>	41 42 43
int	<pre>MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval, void *attribute_val)</pre>	44 45
int	<pre>MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)</pre>	46 47 48

1 2 3	<pre>int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval, void *extra_state)</pre>
4 5	<pre>int MPI_Win_delete_attr(MPI_Win win, int win_keyval)</pre>
6 7 8	<pre>int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,</pre>
9	<pre>int MPI_Win_free_keyval(int *win_keyval)</pre>
10 11 12	<pre>int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>
12	int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
14 15 16	<pre>int 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>
17 18	<pre>int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void</pre>
19 20	int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
21 22	<pre>int MPI_Win_set_name(MPI_Win win, const char *win_name)</pre>
23 24	A.2.5 Process Topologies C Bindings
25	int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
26 27 28	<pre>int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const</pre>
29	<pre>int MPI_Cartdim_get(MPI_Comm comm, int *ndims)</pre>
30 31 32	<pre>int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>
33 34	<pre>int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[], const</pre>
35 36	<pre>int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>
37 38	<pre>int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>
$\frac{39}{40}$	int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
41	int MPI_Dims_create(int nnodes, int ndims, int dims[])
42 43 44 45 46	<pre>int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree, const</pre>
47 48	<pre>int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>

	i	const int degrees[], const int destinations[], const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)	1 2 3
int	-	graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted)	4 5 6
int	i	graph_neighbors(MPI_Comm comm, int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[])	7 8 9 10
int	-	create(MPI_Comm comm_old, int nnodes, const int index[], const int edges[], int reorder, MPI_Comm *comm_graph)	11 12
int	MPI_Graphd	lims_get(MPI_Comm comm, int *nnodes, int *nedges)	13
int	-	<pre>_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[])</pre>	14 15 16
int	-	<pre>map(MPI_Comm comm, int nnodes, const int index[], const int edges[], int *newrank)</pre>	17 18 19
int	MPI_Graph_	neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	20
int	-	_neighbors(MPI_Comm comm, int rank, int maxneighbors, int neighbors[])	21 22 23
int	Ν	abor_allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	24 25 26
int	N	abor_allgatherv(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)	27 28 29 30 31
int	c c	abor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	32 33 34
int		abor_alltoallv(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)	35 36 37 38 39
int	י ז נ	abor_alltoallw(const void* sendbuf, const int sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void* recvbuf, const int recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)	40 41 42 43 44
int	5	oor_allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)	45 46 47 48

1int MPI\_Neighbor\_allgatherv(const void\* sendbuf, int sendcount,  $\mathbf{2}$ MPI\_Datatype sendtype, void\* recvbuf, const int recvcounts[], 3 const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm) 4 int MPI\_Neighbor\_alltoall(const void\* sendbuf, int sendcount, MPI\_Datatype 5sendtype, void\* recvbuf, int recvcount, MPI\_Datatype recvtype, 6 MPI\_Comm comm) 7 8 int MPI\_Neighbor\_alltoallv(const void\* sendbuf, const int sendcounts[], 9 const int sdispls[], MPI\_Datatype sendtype, void\* recvbuf, 10const int recvcounts[], const int rdispls[], MPI\_Datatype 11 recvtype, MPI\_Comm comm) 12int MPI\_Neighbor\_alltoallw(const void\* sendbuf, const int sendcounts[], 13 const MPI\_Aint sdispls[], const MPI\_Datatype sendtypes[], 14 void\* recvbuf, const int recvcounts[], const MPI\_Aint 15rdispls[], const MPI\_Datatype recvtypes[], MPI\_Comm comm) 1617int MPI\_Topo\_test(MPI\_Comm comm, int \*status) 18 19A.2.6 MPI Environmental Management C Bindings 2021int MPI\_Abort(MPI\_Comm comm, int errorcode) 22 int MPI\_Add\_error\_class(int \*errorclass) 23 $^{24}$ int MPI\_Add\_error\_code(int errorclass, int \*errorcode) 2526int MPI\_Add\_error\_string(int errorcode, const char \*string) 27int MPI\_Alloc\_mem(MPI\_Aint size, MPI\_Info info, void \*baseptr) 28 29int MPI\_Comm\_call\_errhandler(MPI\_Comm comm, int errorcode) 30 int MPI\_Comm\_create\_errhandler(MPI\_Comm\_errhandler\_function  $^{31}$ \*comm\_errhandler\_fn, MPI\_Errhandler \*errhandler) 32 33 int MPI\_Comm\_get\_errhandler(MPI\_Comm comm, MPI\_Errhandler \*errhandler) 34 int MPI\_Comm\_set\_errhandler(MPI\_Comm comm, MPI\_Errhandler errhandler) 35 36 int MPI\_Errhandler\_free(MPI\_Errhandler \*errhandler) 37 int MPI\_Error\_class(int errorcode, int \*errorclass) 38 39 int MPI\_Error\_string(int errorcode, char \*string, int \*resultlen) 40int MPI\_File\_call\_errhandler(MPI\_File fh, int errorcode) 41 42int MPI\_File\_create\_errhandler(MPI\_File\_errhandler\_function 43 \*file\_errhandler\_fn, MPI\_Errhandler \*errhandler) 44 int MPI\_File\_get\_errhandler(MPI\_File file, MPI\_Errhandler \*errhandler) 4546int MPI\_File\_set\_errhandler(MPI\_File file, MPI\_Errhandler errhandler) 47int MPI\_Finalized(int \*flag) 48

```
1
int MPI_Finalize(void)
                                                                                    2
int MPI_Free_mem(void *base)
                                                                                    3
                                                                                    4
int MPI_Get_library_version(char *version, int *resultlen)
                                                                                    5
int MPI_Get_processor_name(char *name, int *resultlen)
                                                                                    6
                                                                                    7
int MPI_Get_version(int *version, int *subversion)
int MPI_Initialized(int *flag)
                                                                                    9
                                                                                    10
int MPI_Init(int *argc, char *((*argv)[]))
                                                                                    11
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
                                                                                    12
                                                                                    13
int MPI_Win_create_errhandler(MPI_Win_errhandler_function
                                                                                   14
              *win_errhandler_fn, MPI_Errhandler *errhandler)
                                                                                    15
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
                                                                                    16
                                                                                    17
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
                                                                                    18
double MPI_Wtick(void)
                                                                                    19
                                                                                    20
double MPI_Wtime(void)
                                                                                   21
                                                                                    22
A.2.7 The Info Object C Bindings
                                                                                    23
                                                                                    ^{24}
int MPI_Info_create(MPI_Info *info)
                                                                                    25
                                                                                    26
int MPI_Info_delete(MPI_Info info, const char *key)
                                                                                   27
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
                                                                                   28
                                                                                   29
int MPI_Info_free(MPI_Info *info)
                                                                                    30
int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
                                                                                   31
              int *flag)
                                                                                    32
                                                                                    33
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
                                                                                   34
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
                                                                                   35
                                                                                   36
int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
                                                                                   37
              int *flag)
                                                                                    38
int MPI_Info_set(MPI_Info info, const char *key, const char *value)
                                                                                    39
                                                                                    40
                                                                                    41
A.2.8 Process Creation and Management C Bindings
                                                                                    42
int MPI_Close_port(const char *port_name)
                                                                                    43
                                                                                    44
int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
                                                                                    45
              MPI_Comm comm, MPI_Comm *newcomm)
                                                                                    46
                                                                                    47
int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
                                                                                    48
              MPI_Comm comm, MPI_Comm *newcomm)
```

1	nt MPI_Comm_disconnect(MPI_Comm *comm)	
2 3	nt MPI_Comm_get_parent(MPI_Comm *parent)	
4	nt MPI_Comm_join(int fd, MPI_Comm *intercomm)	
5 6 7 8	nt MPI_Comm_spawn(const char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])	
9 10 11 12 13	<pre>nt MPI_Comm_spawn_multiple(int count, char *array_of_commands[],</pre>	
13 14 15	nt MPI_Lookup_name(const char *service_name, MPI_Info info, char *port_name)	
16 17	nt MPI_Open_port(MPI_Info info, char *port_name)	
18 19	nt MPI_Publish_name(const char *service_name, MPI_Info info, const char *port_name)	
20 21 22 23	nt MPI_Unpublish_name(const char *service_name, MPI_Info info, const char *port_name)	
23	.2.9 One-Sided Communications C Bindings	
25 26 27 28 29	nt MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	
26 27 28	nt MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,	,
26 27 28 29 30 31	nt MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win) nt MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr void *result_addr, MPI_Datatype datatype, int target_rank,	-
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>nt MPI_Accumulate(const void *origin_addr, int origin_count,</pre>	-
26 27 28 30 31 32 33 34 35 36 37 38 39 40	<pre>nt MPI_Accumulate(const void *origin_addr, int origin_count,</pre>	-

0	rigin_datatype, int target_rank, MPI_Aint target_disp, int	1
	arget_count, MPI_Datatype target_datatype, MPI_Win win)	2
int MDT Paccum	ulate(const void *origin_addr, int origin_count,	3
	PI_Datatype origin_datatype, int target_rank,	4
	PI_Aint target_disp, int target_count,	5
	PI_Datatype target_datatype, MPI_Op op, MPI_Win win,	6 7
	PI_Request *request)	8
int MPT Boet ac	ccumulate(const void *origin_addr, int origin_count,	9
-	PI_Datatype origin_datatype, void *result_addr,	10
	nt result_count, MPI_Datatype result_datatype,	11
	nt target_rank, MPI_Aint target_disp, int target_count,	12
M	PI_Datatype target_datatype, MPI_Op op, MPI_Win win,	13
M	PI_Request *request)	14
int MPI_Rget(vo	<pre>pid *origin_addr, int origin_count,</pre>	15 16
-	PI_Datatype origin_datatype, int target_rank,	17
M	PI_Aint target_disp, int target_count,	18
	PI_Datatype target_datatype, MPI_Win win,	19
M	PI_Request *request)	20
int MPI_Rput(co	onst void *origin_addr, int origin_count,	21
M	PI_Datatype origin_datatype, int target_rank,	22
M	PI_Aint target_disp, int target_count,	23
	PI_Datatype target_datatype, MPI_Win win,	24 25
M	PI_Request *request)	25 26
int MPI_Win_all	locate(MPI_Aint size, int disp_unit, MPI_Info info,	27
M	PI_Comm comm, void *baseptr, MPI_Win *win)	28
int MPI_Win_all	locate_shared(MPI_Aint size, int disp_unit, MPI_Info info,	29
	PI_Comm comm, void *baseptr, MPI_Win *win)	30
int MDT Win att	-	31
IIIC MFI_WIII_act	tach(MPI_Win win, void *base, MPI_Aint size)	32 33
int MPI_Win_com	nplete(MPI_Win win)	34
int MPI_Win_cre	eate_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)	35
int MPT Win cre	eate(void *base, MPI_Aint size, int disp_unit, MPI_Info info,	36
	PI_Comm comm, MPI_Win *win)	37
		38
Int MPI_WIN_det	tach(MPI_Win win, const void *base)	39 40
int MPI_Win_fer	nce(int assert, MPI_Win win)	41
int MPI_Win_flu	ush_all(MPI_Win win)	42
int MPI_Win_flu	ush(int rank, MPI_Win win)	43
int MPT Win flu	ush_local_all(MPI_Win win)	44 45
		46
int MP1_Win_flu	ush_local(int rank, MPI_Win win)	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_all(int assert, MPI_Win win)
6
\overline{7}
     int MPI_Win_lock(int lock_type, int rank, 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_all(MPI_Win win)
19
20
     int MPI_Win_unlock(int rank, 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,
33
                   int *provided)
34
     int MPI_Is_thread_main(int *flag)
35
36
     int MPI_Query_thread(int *provided)
37
     int MPI_Status_set_cancelled(MPI_Status *status, int flag)
38
39
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
40
                   int count)
41
     int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
42
                   MPI_Count count)
43
44
45
     A.2.11 I/O C Bindings
46
47
     int MPI_File_close(MPI_File *fh)
48
```

int	MPI_File_delete(const char *filename, MPI_Info info)	1
int	MPI_File_get_amode(MPI_File fh, int *amode)	2 3
int	MPI_File_get_atomicity(MPI_File fh, int *flag)	4
		5
int	<pre>MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>	6
int.	•	7 8
	MPI_File_get_group(MPI_File fh, MPI_Group *group)	9
int	MPI_File_get_info(MPI_File fh, MPI_Info *info_used)	10
int	MPI_File_get_position(MPI_File fh, MPI_Offset *offset)	11 12
int	MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)	13
int	MPI_File_get_size(MPI_File fh, MPI_Offset *size)	14
int	MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,	15 16
1110	MPI_Aint *extent)	17
int	MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,	18
1110	MPI_Datatype *filetype, char *datarep)	19
int	MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,	20 21
1110	MPI_Datatype datatype, MPI_Request *request)	22
int	MPI_File_iread(MPI_File fh, void *buf, int count,	23
	MPI_Datatype datatype, MPI_Request *request)	24
int	MPI_File_iread_shared(MPI_File fh, void *buf, int count,	25 26
1110	MPI_Datatype datatype, MPI_Request *request)	27
int	MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,	28
1110	int count, MPI_Datatype datatype, MPI_Request *request)	29 30
int	MPI_File_iwrite(MPI_File fh, const void *buf, int count,	31
1110	MPI_Datatype datatype, MPI_Request *request)	32
int	MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,	33
	MPI_Datatype datatype, MPI_Request *request)	$\frac{34}{35}$
int	MPI_File_open(MPI_Comm comm, const char *filename, int amode,	36
	MPI_Info info, MPI_File *fh)	37
int	MPI_File_preallocate(MPI_File fh, MPI_Offset size)	38
	•	39 40
THC	<pre>MPI_File_read_all_begin(MPI_File fh, void *buf, int count,</pre>	41
in+	MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	42
		43 44
int	<pre>MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	45
• •		46
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	47
	ind count, in i_basabype addabype?	48

## ANNEX A. LANGUAGE BINDINGS SUMMARY

1 int MPI\_File\_read\_at\_all\_end(MPI\_File fh, void \*buf, MPI\_Status \*status)  $\mathbf{2}$ int MPI\_File\_read\_at\_all(MPI\_File fh, MPI\_Offset offset, void \*buf, 3 int count, MPI\_Datatype datatype, MPI\_Status \*status) 4  $\mathbf{5}$ int MPI\_File\_read\_at(MPI\_File fh, MPI\_Offset offset, void \*buf, int count, 6 MPI\_Datatype datatype, MPI\_Status \*status) 7int MPI\_File\_read(MPI\_File fh, void \*buf, int count, MPI\_Datatype datatype, 8 MPI\_Status \*status) 9 10int MPI\_File\_read\_ordered\_begin(MPI\_File fh, void \*buf, int count,  $^{11}$ MPI\_Datatype datatype) 12int MPI\_File\_read\_ordered\_end(MPI\_File fh, void \*buf, MPI\_Status \*status) 1314int MPI\_File\_read\_ordered(MPI\_File fh, void \*buf, int count, 15MPI\_Datatype datatype, MPI\_Status \*status) 16int MPI\_File\_read\_shared(MPI\_File fh, void \*buf, int count, 17MPI\_Datatype datatype, MPI\_Status \*status) 18 19int MPI\_File\_seek(MPI\_File fh, MPI\_Offset offset, int whence) 20int MPI\_File\_seek\_shared(MPI\_File fh, MPI\_Offset offset, int whence) 2122int MPI\_File\_set\_atomicity(MPI\_File fh, int flag) 23int MPI\_File\_set\_info(MPI\_File fh, MPI\_Info info)  $^{24}$ 25int MPI\_File\_set\_size(MPI\_File fh, MPI\_Offset size) 26int MPI\_File\_set\_view(MPI\_File fh, MPI\_Offset disp, MPI\_Datatype etype, 27MPI\_Datatype filetype, const char \*datarep, MPI\_Info info) 2829int MPI\_File\_sync(MPI\_File fh) 30  $^{31}$ int MPI\_File\_write\_all\_begin(MPI\_File fh, const void \*buf, int count, 32MPI\_Datatype datatype) 33 int MPI\_File\_write\_all\_end(MPI\_File fh, const void \*buf, 34 MPI\_Status \*status) 3536 int MPI\_File\_write\_all(MPI\_File fh, const void \*buf, int count, 37 MPI\_Datatype datatype, MPI\_Status \*status) 38int MPI\_File\_write\_at\_all\_begin(MPI\_File fh, MPI\_Offset offset, const 39 void \*buf, int count, MPI\_Datatype datatype) 4041int MPI\_File\_write\_at\_all\_end(MPI\_File fh, const void \*buf, 42MPI\_Status \*status) 43 int MPI\_File\_write\_at\_all(MPI\_File fh, MPI\_Offset offset, const void \*buf, 44int count, MPI\_Datatype datatype, MPI\_Status \*status) 4546int MPI\_File\_write\_at(MPI\_File fh, MPI\_Offset offset, const void \*buf, 47int count, MPI\_Datatype datatype, MPI\_Status \*status)

48

<pre>int MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	1 2
<pre>int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>	3 4 5
<pre>int MPI_File_write_ordered_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>	6 7
<pre>int MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	8 9 10
<pre>int MPI_File_write_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	11 12
<pre>int 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
<pre>int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)</pre>	25
<pre>int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)</pre>	26 27
<pre>int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)</pre>	27
<pre>int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)</pre>	29 30
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)	30
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)	32
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	33 34
	35
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	36
MPI_Fint MPI_File_c2f(MPI_File file)	37 38
MPI_File MPI_File_f2c(MPI_Fint file)	39
MPI_Fint MPI_Group_c2f(MPI_Group group)	40
MPI_Group MPI_Group_f2c(MPI_Fint group)	41 42
MPI_Fint MPI_Info_c2f(MPI_Info info)	43
	44
MPI_Info MPI_Info_f2c(MPI_Fint info)	45 46
MPI_Fint MPI_Message_c2f(MPI_Message message)	47
MPI_Message MPI_Message_f2c(MPI_Fint message)	48

```
1
     MPI_Fint MPI_Op_c2f(MPI_Op op)
\mathbf{2}
     MPI_Op MPI_Op_f2c(MPI_Fint op)
3
4
     MPI_Fint MPI_Request_c2f(MPI_Request request)
5
     MPI_Request MPI_Request_f2c(MPI_Fint request)
6
7
     int MPI_Status_c2f08(const MPI_Status *c_status, MPI_F08_status
8
                   *f08_status)
9
     int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)
10
11
     int MPI_Status_f082c(const MPI_F08_status *f08_status, MPI_Status
12
                   *c status)
13
     int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
14
15
     MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
16
     MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
17
18
     MPI_Fint MPI_Win_c2f(MPI_Win win)
19
     MPI_Win MPI_Win_f2c(MPI_Fint win)
20
21
22
     A.2.13 Tools / Profiling Interface C Bindings
23
^{24}
     int MPI_Pcontrol(const int level, ...)
25
26
     A.2.14 Tools / MPI Tool Information Interface C Bindings
27
28
     int MPI_T_category_changed(int *stamp)
29
     int MPI_T_category_get_categories(int cat_index, int len, int indices[])
30
^{31}
     int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
32
     int MPI_T_category_get_info(int cat_index, char *name, int *name_len, char
33
34
                   *desc, int *desc_len, int *num_cvars, int *num_pvars, int
                   *num_categories)
35
36
     int MPI_T_category_get_num(int *num_cat)
37
38
     int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
39
     int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len, int
40
                   *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype, char
41
                   *desc, int *desc_len, int *bind, int *scope)
42
43
     int MPI_T_cvar_get_num(int *num_cvar)
44
     int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
45
                   MPI_T_cvar_handle *handle, int *count)
46
47
     int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
48
```

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
int	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len, int</pre>	12 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	<pre>MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>	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
	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
	void *attribute_val_in, void *attribute_val_out, int *flag)	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) int MPI\_NULL\_COPY\_FN(MPI\_Comm oldcomm, int keyval, void \*extra\_state, void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag) int MPI\_NULL\_DELETE\_FN(MPI\_Comm comm, int keyval, void \*attribute\_val, void \*extra\_state)  $^{24}$ 

A.3. FORTRAN 2008 BINDINGS WITH THE MPI_F08 MODULE	709
A.3 Fortran 2008 Bindings with the mpi_f08 Module	1
A 2.1 Drive to Drive Communication Frature 2000 Divisions	2
A.3.1 Point-to-Point Communication Fortran 2008 Bindings	3
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)	4 5
TYPE(*), DIMENSION(), INTENT(IN) :: buf	5
INTEGER, INTENT(IN) :: count, dest, tag	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)	11
BIND(C) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	12 13
INTEGER, INTENT(IN) :: count, dest, tag	13
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16
TYPE(MPI_Request), INTENT(OUT) :: request	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
NDT Duffer attack (buffer airs issues) DIND(C)	19
<pre>MPI_Buffer_attach(buffer, size, ierror) BIND(C)     TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer</pre>	20
INTEGER, INTENT(IN) :: size	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	23
MPI_Buffer_detach(buffer_addr, size, ierror) BIND(C)	24
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	25 26
TYPE(C_PTR), INTENT(OUT) :: buffer_addr INTEGER, INTENT(OUT) :: size	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
	29
MPI_Cancel(request, ierror) BIND(C)	30
TYPE(MPI_Request), INTENT(IN) :: request	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
MPI_Get_count(status, datatype, count, ierror) BIND(C)	33
TYPE(MPI_Status), INTENT(IN) :: status	34
TYPE(MPI_Datatype), INTENT(IN) :: datatype	35
INTEGER, INTENT(OUT) :: count	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37 38
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(	
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	40
INTEGER, INTENT(IN) :: count, dest, tag	41
TYPE(MPI_Datatype), INTENT(IN) :: datatype	42
TYPE(MPI_Comm), INTENT(IN) :: comm	43
TYPE(MPI_Request), INTENT(OUT) :: request	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
MPI_Improbe(source, tag, comm, flag, message, status, ierror) BIND(C)	46
INTEGER, INTENT(IN) :: source, tag	47
	48

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, INTENT(OUT) :: flag
3
         TYPE(MPI_Message), INTENT(OUT) :: message
4
         TYPE(MPI_Status) :: status
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
    MPI_Imrecv(buf, count, datatype, message, request, ierror) BIND(C)
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
8
         INTEGER, INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Message), INTENT(INOUT) :: message
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
    MPI_Iprobe(source, tag, comm, flag, status, ierror) BIND(C)
15
         INTEGER, INTENT(IN) :: source, tag
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         LOGICAL, INTENT(OUT) :: flag
18
         TYPE(MPI_Status) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
    MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND(C)
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count, source, tag
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
29
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
30
         INTEGER, INTENT(IN) :: count, dest, tag
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
    MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
36
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count, dest, tag
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
44
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
45
         INTEGER, INTENT(IN) :: count, dest, tag
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 3
MPI_Mprobe(source, tag, comm, message, status, ierror) BIND(C)
                                                                                 4
    INTEGER, INTENT(IN) :: source, tag
                                                                                 5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 6
    TYPE(MPI_Message), INTENT(OUT) :: message
                                                                                 7
    TYPE(MPI_Status) :: status
                                                                                 8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 9
                                                                                 10
MPI_Mrecv(buf, count, datatype, message, status, ierror) BIND(C)
                                                                                 11
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
                                                                                 12
                                                                                 13
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 14
    TYPE(MPI_Message), INTENT(INOUT) :: message
                                                                                 15
    TYPE(MPI_Status) :: status
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 17
MPI_Probe(source, tag, comm, status, ierror) BIND(C)
                                                                                 18
    INTEGER, INTENT(IN) :: source, tag
                                                                                 19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 20
    TYPE(MPI_Status) :: status
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 22
                                                                                 23
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C)
                                                                                 24
    TYPE(*), DIMENSION(..) :: buf
                                                                                 25
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                 26
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 28
    TYPE(MPI_Status) :: status
                                                                                 29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 30
MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
                                                                                 31
             BIND(C)
                                                                                 32
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 33
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                 34
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 36
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 38
                                                                                 39
MPI_Request_free(request, ierror) BIND(C)
                                                                                 40
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
MPI_Request_get_status(request, flag, status, ierror) BIND(C)
                                                                                 43
    TYPE(MPI_Request), INTENT(IN) :: request
                                                                                 44
    LOGICAL, INTENT(OUT) :: flag
                                                                                 45
    TYPE(MPI_Status) :: status
                                                                                 46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 47
                                                                                 48
```

```
1
    MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
\mathbf{2}
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
3
         INTEGER, INTENT(IN) :: count, dest, tag
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         TYPE(MPI_Comm), INTENT(IN) :: comm
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
8
                  BIND(C)
9
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
10
         INTEGER, INTENT(IN) :: count, dest, tag
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
17
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
18
         INTEGER, INTENT(IN) :: count, dest, tag
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
23
                  BIND(C)
24
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
25
         INTEGER, INTENT(IN) :: count, dest, tag
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Request), INTENT(OUT) :: request
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
32
                  comm, status, ierror) BIND(C)
33
         TYPE(*), DIMENSION(..) :: buf
34
         INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Status) :: status
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
40
                  recvcount, recvtype, source, recvtag, comm, status, ierror)
41
                  BIND(C)
42
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
43
         TYPE(*), DIMENSION(..) :: recvbuf
44
         INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
45
         recvtag
46
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

1 TYPE(MPI\_Status) :: status 2 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3 MPI\_Ssend(buf, count, datatype, dest, tag, comm, ierror) BIND(C) 4 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 5 INTEGER, INTENT(IN) :: count, dest, tag 6 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI\_Comm), INTENT(IN) :: comm 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 MPI\_Ssend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 11 BIND(C) TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 1213 INTEGER, INTENT(IN) :: count, dest, tag 14TYPE(MPI\_Datatype), INTENT(IN) :: datatype 15TYPE(MPI\_Comm), INTENT(IN) :: comm 16TYPE(MPI\_Request), INTENT(OUT) :: request 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI\_Startall(count, array\_of\_requests, ierror) BIND(C) 19 INTEGER, INTENT(IN) :: count 20TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI\_Start(request, ierror) BIND(C) 24TYPE(MPI\_Request), INTENT(INOUT) :: request 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI\_Testall(count, array\_of\_requests, flag, array\_of\_statuses, ierror) 27BIND(C) 28 INTEGER, INTENT(IN) :: count 29 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 30 LOGICAL, INTENT(OUT) :: flag 31TYPE(MPI\_Status) :: array\_of\_statuses(\*) 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 34 MPI\_Testany(count, array\_of\_requests, index, flag, status, ierror) BIND(C) 35 INTEGER, INTENT(IN) :: count 36 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 37 INTEGER, INTENT(OUT) :: index 38 LOGICAL, INTENT(OUT) :: flag 39 TYPE(MPI\_Status) :: status 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI\_Test\_cancelled(status, flag, ierror) BIND(C) 42TYPE(MPI\_Status), INTENT(IN) :: status 43 LOGICAL, INTENT(OUT) :: flag 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546MPI\_Test(request, flag, status, ierror) BIND(C) 47TYPE(MPI\_Request), INTENT(INOUT) :: request 48

```
1
         LOGICAL, INTENT(OUT) :: flag
2
         TYPE(MPI_Status) :: status
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
5
                  array_of_statuses, ierror) BIND(C)
6
         INTEGER, INTENT(IN) :: incount
7
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
8
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
9
         TYPE(MPI_Status) :: array_of_statuses(*)
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) BIND(C)
13
         INTEGER, INTENT(IN) :: count
14
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
15
         TYPE(MPI_Status) :: array_of_statuses(*)
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Waitany(count, array_of_requests, index, status, ierror) BIND(C)
18
         INTEGER, INTENT(IN) :: count
19
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
20
         INTEGER, INTENT(OUT) :: index
21
         TYPE(MPI_Status) :: status
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
    MPI_Wait(request, status, ierror) BIND(C)
25
         TYPE(MPI_Request), INTENT(INOUT) :: request
26
         TYPE(MPI_Status) :: status
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
29
                  array_of_statuses, ierror) BIND(C)
30
         INTEGER, INTENT(IN) :: incount
31
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
32
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
33
         TYPE(MPI_Status) :: array_of_statuses(*)
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
37
     A.3.2 Datatypes Fortran 2008 Bindings
38
    MPI_Get_address(location, address, ierror) BIND(C)
39
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: location
40
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_Get_elements(status, datatype, count, ierror) BIND(C)
44
         TYPE(MPI_Status), INTENT(IN) :: status
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         INTEGER, INTENT(OUT) :: count
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

1 MPI\_Get\_elements\_x(status, datatype, count, ierror) BIND(C) 2 TYPE(MPI\_Status), INTENT(IN) :: status 3 TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER(KIND = MPI\_COUNT\_KIND), INTENT(OUT) :: count 4 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 56 MPI\_Pack\_external(datarep, inbuf, incount, datatype, outbuf, outsize, 7 position, ierror) BIND(C) CHARACTER(LEN=\*), INTENT(IN) :: datarep 9 TYPE(\*), DIMENSION(..), INTENT(IN) :: inbuf 10 TYPE(\*), DIMENSION(..) :: outbuf 11 INTEGER, INTENT(IN) :: incount 12TYPE(MPI\_Datatype), INTENT(IN) :: datatype 13 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: outsize 14INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(INOUT) :: position 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617MPI\_Pack\_external\_size(datarep, incount, datatype, size, ierror) BIND(C) 18 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 19 INTEGER, INTENT(IN) :: incount CHARACTER(LEN=\*), INTENT(IN) :: datarep 2021INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: size 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23MPI\_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror) 24BIND(C) 25TYPE(\*), DIMENSION(..), INTENT(IN) :: inbuf 26TYPE(\*), DIMENSION(..) :: outbuf 27INTEGER, INTENT(IN) :: incount, outsize 28 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 29 INTEGER, INTENT(INOUT) :: position 30 TYPE(MPI\_Comm), INTENT(IN) :: comm 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 MPI\_Pack\_size(incount, datatype, comm, size, ierror) BIND(C) 34 INTEGER, INTENT(IN) :: incount 35 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI\_Comm), INTENT(IN) :: comm 37 INTEGER, INTENT(OUT) :: size 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI\_Type\_commit(datatype, ierror) BIND(C) 40 TYPE(MPI\_Datatype), INTENT(INOUT) :: datatype 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI\_Type\_contiguous(count, oldtype, newtype, ierror) BIND(C) 44INTEGER, INTENT(IN) :: count 45TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 46TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
1
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
2
                  array_of_distribs, array_of_dargs, array_of_psizes, order,
3
                  oldtype, newtype, ierror) BIND(C)
4
         INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
         array_of_distribs(ndims), array_of_dargs(ndims),
5
6
         array_of_psizes(ndims), order
7
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
8
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
11
                  oldtype, newtype, ierror) BIND(C)
12
         INTEGER, INTENT(IN) :: count, blocklength
13
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
14
         array_of_displacements(count)
15
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
16
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Type_create_hindexed(count, array_of_blocklengths,
20
                  array_of_displacements, oldtype, newtype, ierror) BIND(C)
21
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
22
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
23
         array_of_displacements(count)
24
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
25
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
28
                  ierror) BIND(C)
29
         INTEGER, INTENT(IN) :: count, blocklength
30
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                         stride
31
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
32
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
36
                  oldtype, newtype, ierror) BIND(C)
37
         INTEGER, INTENT(IN) :: count, blocklength,
38
         array_of_displacements(count)
39
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
40
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) BIND(C)
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
44
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
45
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

1 MPI\_Type\_create\_struct(count, array\_of\_blocklengths, 2 array\_of\_displacements, array\_of\_types, newtype, ierror) BIND(C) 4 INTEGER, INTENT(IN) :: count, array\_of\_blocklengths(count) INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: 5 array\_of\_displacements(count) 6 7 TYPE(MPI\_Datatype), INTENT(IN) :: array\_of\_types(count) TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10MPI\_Type\_create\_subarray(ndims, array\_of\_sizes, array\_of\_subsizes, 11 array\_of\_starts, order, oldtype, newtype, ierror) BIND(C) 12INTEGER, INTENT(IN) :: ndims, array\_of\_sizes(ndims), 13 array\_of\_subsizes(ndims), array\_of\_starts(ndims), order 14 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 15TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 MPI\_Type\_dup(oldtype, newtype, ierror) BIND(C) 19 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 2021INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 MPI\_Type\_free(datatype, ierror) BIND(C) 23TYPE(MPI\_Datatype), INTENT(INOUT) :: datatype 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI\_Type\_get\_contents(datatype, max\_integers, max\_addresses, max\_datatypes, 27array\_of\_integers, array\_of\_addresses, array\_of\_datatypes, 28 ierror) BIND(C) 29 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 30 INTEGER, INTENT(IN) :: max\_integers, max\_addresses, max\_datatypes 31INTEGER, INTENT(OUT) :: array\_of\_integers(max\_integers) 32 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: 33 array\_of\_addresses(max\_addresses) 34 TYPE(MPI\_Datatype), INTENT(OUT) :: array\_of\_datatypes(max\_datatypes) 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI\_Type\_get\_envelope(datatype, num\_integers, num\_addresses, num\_datatypes, 37 combiner, ierror) BIND(C) 38 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 39 INTEGER, INTENT(OUT) :: num\_integers, num\_addresses, num\_datatypes, 40 combiner 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI\_Type\_get\_extent(datatype, lb, extent, ierror) BIND(C) 44 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 45INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: lb, extent 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47MPI\_Type\_get\_extent\_x(datatype, lb, extent, ierror) BIND(C) 48

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
\mathbf{2}
         INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) BIND(C)
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror) BIND(C)
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
14
                  oldtype, newtype, ierror) BIND(C)
15
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
16
         array_of_displacements(count)
17
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
18
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Type_size(datatype, size, ierror) BIND(C)
22
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
         INTEGER, INTENT(OUT) :: size
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Type_size_x(datatype, size, ierror) BIND(C)
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
^{31}
                  BIND(C)
32
         INTEGER, INTENT(IN) :: count, blocklength, stride
33
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
34
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
37
                  datatype, ierror) BIND(C)
38
         CHARACTER(LEN=*), INTENT(IN) :: datarep
39
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
40
         TYPE(*), DIMENSION(..) :: outbuf
41
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
43
         INTEGER, INTENT(IN) :: outcount
44
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
48
```

```
1
             ierror) BIND(C)
                                                                                 2
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
                                                                                 3
    TYPE(*), DIMENSION(..) :: outbuf
    INTEGER, INTENT(IN) :: insize, outcount
                                                                                 4
    INTEGER, INTENT(INOUT) :: position
                                                                                 5
                                                                                 6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 7
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 8
                                                                                 9
                                                                                 10
A.3.3 Collective Communication Fortran 2008 Bindings
                                                                                 11
                                                                                 12
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 13
             comm, ierror) BIND(C)
                                                                                 14
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 15
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 16
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 17
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 18
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 20
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                 21
             recvtype, comm, ierror) BIND(C)
                                                                                 22
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 23
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 24
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                 25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 26
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 28
                                                                                 29
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
                                                                                 30
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 31
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 32
    INTEGER, INTENT(IN) :: count
                                                                                 33
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 34
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 37
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 38
             comm, ierror) BIND(C)
                                                                                 39
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 40
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 41
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 42
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 45
                                                                                 46
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                 47
             rdispls, recvtype, comm, ierror) BIND(C)
                                                                                 48
```

```
1
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
2
         TYPE(*), DIMENSION(..) :: recvbuf
3
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
4
         rdispls(*)
5
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
    MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
9
                  rdispls, recvtypes, comm, ierror) BIND(C)
10
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
11
         TYPE(*), DIMENSION(..) :: recvbuf
12
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
13
         rdispls(*)
14
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
15
         TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Barrier(comm, ierror) BIND(C)
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
    MPI_Bcast(buffer, count, datatype, root, comm, ierror) BIND(C)
23
         TYPE(*), DIMENSION(..) :: buffer
24
         INTEGER, INTENT(IN) :: count, root
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
    MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
30
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
31
         TYPE(*), DIMENSION(..) :: recvbuf
32
         INTEGER, INTENT(IN) :: count
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Op), INTENT(IN) :: op
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
    MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
38
                  root, comm, ierror) BIND(C)
39
         TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                sendbuf
40
         TYPE(*), DIMENSION(..) :: recvbuf
41
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
42
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
47
                  recvtype, root, comm, ierror) BIND(C)
48
```

```
TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 1
                                                                                 2
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 3
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 4
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 6
                                                                                 7
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 8
             comm, request, ierror) BIND(C)
                                                                                 9
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 10
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                 11
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                12
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                14
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 16
                                                                                17
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                18
             recvtype, comm, request, ierror) BIND(C)
                                                                                19
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                20
                                                                                21
    INTEGER, INTENT(IN) :: sendcount
                                                                                22
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                23
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                27
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
                                                                                28
             ierror) BIND(C)
                                                                                29
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                30
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 31
    INTEGER, INTENT(IN) :: count
                                                                                 32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                33
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                35
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                37
                                                                                38
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                39
             comm, request, ierror) BIND(C)
                                                                                 40
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                41
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                42
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                43
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 45
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 47
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                48
```

1 rdispls, recvtype, comm, request, ierror) BIND(C) 2 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 3 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 4 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(\*), sdispls(\*), 5recvcounts(\*), rdispls(\*) 6 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 7 TYPE(MPI\_Comm), INTENT(IN) :: comm 8 TYPE(MPI\_Request), INTENT(OUT) :: request 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 MPI\_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, 11 recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C) 12TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 13 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 14 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(\*), sdispls(\*), 15recvcounts(\*), rdispls(\*) 16 TYPE(MPI\_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(\*), 17 recvtypes(\*) 18 TYPE(MPI\_Comm), INTENT(IN) :: comm 19 TYPE(MPI\_Request), INTENT(OUT) :: request 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI\_Ibarrier(comm, request, ierror) BIND(C) 23TYPE(MPI\_Comm), INTENT(IN) :: comm 24TYPE(MPI\_Request), INTENT(OUT) :: request 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI\_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C) 27TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buffer 28INTEGER, INTENT(IN) :: count, root 29 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 30 TYPE(MPI\_Comm), INTENT(IN) :: comm 31TYPE(MPI\_Request), INTENT(OUT) :: request 32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 33 34MPI\_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) 35 BIND(C) 36 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 37 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 38 INTEGER, INTENT(IN) :: count 39 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 40TYPE(MPI\_Op), INTENT(IN) :: op 41 TYPE(MPI\_Comm), INTENT(IN) :: comm 42TYPE(MPI\_Request), INTENT(OUT) :: request 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 MPI\_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 45 root, comm, request, ierror) BIND(C) 46TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 47 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 48

```
1
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                 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_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                 7
             recvtype, root, comm, request, ierror) BIND(C)
                                                                                 8
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                10
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                11
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                12
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                14
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 16
                                                                                17
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                18
             request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                20
                                                                                21
    INTEGER, INTENT(IN) :: recvcount
                                                                                22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                23
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                27
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                28
             request, ierror) BIND(C)
                                                                                29
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                30
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                31
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                33
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                35
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                37
                                                                                38
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                39
             ierror) BIND(C)
                                                                                 40
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                41
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                42
    INTEGER, INTENT(IN) :: count, root
                                                                                43
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 44
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 46
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 48
```

1MPI\_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) 2 BIND(C) 3 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 4 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 5INTEGER, INTENT(IN) :: count 6 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI\_Op), INTENT(IN) :: op 8 TYPE(MPI\_Comm), INTENT(IN) :: comm 9 TYPE(MPI\_Request), INTENT(OUT) :: request 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI\_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 12root, comm, request, ierror) BIND(C) 13 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 14 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 15INTEGER, INTENT(IN) :: sendcount, recvcount, root 16 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 17 TYPE(MPI\_Comm), INTENT(IN) :: comm 18 TYPE(MPI\_Request), INTENT(OUT) :: request 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 22 recvtype, root, comm, request, ierror) BIND(C) 23TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 24TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 25INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(\*), displs(\*) 26INTEGER, INTENT(IN) :: recvcount, root 27TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 28TYPE(MPI\_Comm), INTENT(IN) :: comm 29TYPE(MPI\_Request), INTENT(OUT) :: request 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror  $^{31}$ MPI\_Op\_commutative(op, commute, ierror) BIND(C) 32 TYPE(MPI\_Op), INTENT(IN) :: op 33 LOGICAL, INTENT(OUT) :: commute 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 MPI\_Op\_create(user\_fn, commute, op, ierror) BIND(C) 37 PROCEDURE(MPI\_User\_function) :: user\_fn 38 LOGICAL, INTENT(IN) :: commute 39 TYPE(MPI\_Op), INTENT(OUT) :: op 40INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI\_Op\_free(op, ierror) BIND(C) 42TYPE(MPI\_Op), INTENT(INOUT) :: op 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 45MPI\_Reduce\_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C) 46TYPE(\*), DIMENSION(..), INTENT(IN) :: inbuf 47 TYPE(\*), DIMENSION(..) :: inoutbuf 48

```
1
    INTEGER, INTENT(IN) :: count
                                                                                 2
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
                                                                                 5
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                 6
             ierror) BIND(C)
                                                                                 7
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 8
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 9
    INTEGER, INTENT(IN) :: recvcount
                                                                                 10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 11
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 14
                                                                                 15
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                 16
             ierror) BIND(C)
                                                                                 17
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 18
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 19
    INTEGER, INTENT(IN) :: recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 20
                                                                                 21
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 24
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
                                                                                 25
             BIND(C)
                                                                                 26
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 27
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 28
    INTEGER, INTENT(IN) :: count, root
                                                                                 29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 30
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 33
                                                                                 34
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
                                                                                 35
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 36
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 37
    INTEGER, INTENT(IN) :: count
                                                                                 38
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 39
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 43
             root, comm, ierror) BIND(C)
                                                                                 44
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 45
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 46
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                 47
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
    MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
4
                  recvtype, root, comm, ierror) BIND(C)
5
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
6
         TYPE(*), DIMENSION(..) :: recvbuf
7
         INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
13
     A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
14
    MPI_Comm_compare(comm1, comm2, result, ierror) BIND(C)
15
         TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
16
         INTEGER, INTENT(OUT) :: result
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
    MPI_Comm_create(comm, group, newcomm, ierror) BIND(C)
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         TYPE(MPI_Group), INTENT(IN) :: group
22
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
    MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Group), INTENT(IN) :: group
27
         INTEGER, INTENT(IN) :: tag
28
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
    MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
32
                  extra_state, ierror) BIND(C)
33
         PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
34
         PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
35
         INTEGER, INTENT(OUT) :: comm_keyval
36
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
    MPI_Comm_delete_attr(comm, comm_keyval, ierror) BIND(C)
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         INTEGER, INTENT(IN) :: comm_keyval
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_Comm_dup(comm, newcomm, ierror) BIND(C)
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

<pre>MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,</pre>	1
attribute_val_out, flag, ierror) BIND(C)	2
TYPE(MPI_Comm), INTENT(IN) :: oldcomm	3
INTEGER, INTENT(IN) :: comm_keyval	4
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,</pre>	5
attribute_val_in	6
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out	7
LOGICAL, INTENT(OUT) :: flag	8
INTEGER, INTENT(OUT) :: ierror	9
MPI_Comm_dup_with_info(comm, info, newcomm, ierror) BIND(C)	10
TYPE(MPI_Comm), INTENT(IN) :: comm	11
TYPE(MPI_Info), INTENT(IN) :: info	12
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
	15
MPI_Comm_free(comm, ierror) BIND(C)	16
TYPE(MPI_Comm), INTENT(INOUT) :: comm	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)	19 20
INTEGER, INTENT(INOUT) :: comm_keyval	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
	23
MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror) BIND(C)	24
TYPE(MPI_Comm), INTENT(IN) :: comm	25
INTEGER, INTENT(IN) :: comm_keyval	26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val	27
LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
INTEGER, OFITOMAL, INTENT(001) TETTOT	29
<pre>MPI_Comm_get_info(comm, info_used, ierror) BIND(C)</pre>	30
TYPE(MPI_Comm), INTENT(IN) :: comm	31
TYPE(MPI_Info), INTENT(OUT) :: info_used	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
MPI_Comm_get_name(comm, comm_name, resultlen, ierror) BIND(C)	34
TYPE(MPI_Comm), INTENT(IN) :: comm	35
CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name	36
INTEGER, INTENT(OUT) :: resultlen	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
	39
MPI_Comm_group(comm, group, ierror) BIND(C)	40
TYPE(MPI_Comm), INTENT(IN) :: comm	41
TYPE(MPI_Group), INTENT(OUT) :: group	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
MPI_Comm_idup(comm, newcomm, request, ierror) BIND(C)	44
TYPE(MPI_Comm), INTENT(IN) :: comm	45
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	46
TYPE(MPI_Request), INTENT(OUT) :: request	47
	48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
3
                  attribute_val_out, flag, ierror) BIND(C)
4
         TYPE(MPI_Comm), INTENT(IN) :: oldcomm
5
         INTEGER, INTENT(IN) :: comm_keyval
6
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
7
         attribute_val_in
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
9
         LOGICAL, INTENT(OUT) :: flag
10
         INTEGER, INTENT(OUT) :: ierror
11
12
     MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
13
                  ierror) BIND(C)
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         INTEGER, INTENT(IN) :: comm_keyval
16
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
17
         extra_state
18
         INTEGER, INTENT(OUT) :: ierror
19
     MPI_Comm_rank(comm, rank, ierror) BIND(C)
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         INTEGER, INTENT(OUT) :: rank
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Comm_remote_group(comm, group, ierror) BIND(C)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Group), INTENT(OUT) :: group
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Comm_remote_size(comm, size, ierror) BIND(C)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, INTENT(OUT) :: size
^{31}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) BIND(C)
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         INTEGER, INTENT(IN) :: comm_keyval
36
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Comm_set_info(MPI_Comm comm, MPI_Info info) BIND(C)
39
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
40
         TYPE(MPI_Info), INTENT(IN) :: info
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_Comm_set_name(comm, comm_name, ierror) BIND(C)
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         CHARACTER(LEN=*), INTENT(IN) :: comm_name
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_Comm_size(comm, size, ierror) BIND(C)
48
```

TYPE(MPI_Comm), INTENT(IN) :: comm	1 2
INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
MPI_Comm_split(comm, color, key, newcomm, ierror) BIND(C)	4
TYPE(MPI_Comm), INTENT(IN) :: comm	5
INTEGER, INTENT(IN) :: color, key	6
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
	9
<pre>MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) BIND(C)</pre>	10 11
TYPE(MPI_Comm), INTENT(IN) :: comm	11
INTEGER, INTENT(IN) :: split_type, key	13
TYPE(MPI_Info), INTENT(IN) :: info	14
TYPE(MPI_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15
INTEGER, OFITONAL, INTENT(OUT) TETTOT	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
MPI_Group_compare(group1, group2, result, ierror) BIND(C)	21
TYPE(MPI_Group), INTENT(IN) :: group1, group2	22
INTEGER, INTENT(OUT) :: result	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	25
MPI_Group_difference(group1, group2, newgroup, ierror) BIND(C)	26
TYPE(MPI_Group), INTENT(IN) :: group1, group2	27
TYPE(MPI_Group), INTENT(OUT) :: newgroup	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C)	30
TYPE(MPI_Group), INTENT(IN) :: group	31
<pre>INTEGER, INTENT(IN) :: n, ranks(n)</pre>	32
TYPE(MPI_Group), INTENT(OUT) :: newgroup	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34 35
MPI_Group_free(group, ierror) BIND(C)	36
TYPE(MPI_Group), INTENT(INOUT) :: group	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
	39
MPI_Group_incl(group, n, ranks, newgroup, ierror) BIND(C)	40
TYPE(MPI_Group), INTENT(IN) :: group	41
INTEGER, INTENT(IN) :: n, ranks(n)	42
TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
INIEGER, UFILUNAL, INIENI(UUI) :: 101101	44
MPI_Group_intersection(group1, group2, newgroup, ierror) BIND(C)	45
TYPE(MPI_Group), INTENT(IN) :: group1, group2	46
TYPE(MPI_Group), INTENT(OUT) :: newgroup	47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
    MPI_Group_range_excl(group, n, ranges, newgroup, ierror) BIND(C)
\mathbf{2}
         TYPE(MPI_Group), INTENT(IN) :: group
3
         INTEGER, INTENT(IN) :: n, ranges(3,n)
4
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)
7
         TYPE(MPI_Group), INTENT(IN) :: group
8
         INTEGER, INTENT(IN) :: n, ranges(3,n)
9
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Group_rank(group, rank, ierror) BIND(C)
13
         TYPE(MPI_Group), INTENT(IN) :: group
14
         INTEGER, INTENT(OUT) :: rank
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Group_size(group, size, ierror) BIND(C)
17
         TYPE(MPI_Group), INTENT(IN) :: group
18
         INTEGER, INTENT(OUT) :: size
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
22
                  BIND(C)
23
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
24
         INTEGER, INTENT(IN) :: n, ranks1(n)
25
         INTEGER, INTENT(OUT) :: ranks2(n)
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Group_union(group1, group2, newgroup, ierror) BIND(C)
28
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
29
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
33
                  tag, newintercomm, ierror) BIND(C)
34
         TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
35
         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
36
         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) BIND(C)
39
         TYPE(MPI_Comm), INTENT(IN) :: intercomm
40
         LOGICAL, INTENT(IN) :: high
41
         TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
45
                  extra_state, ierror) BIND(C)
46
         PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
47
         PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
48
```

1 INTEGER, INTENT(OUT) :: type\_keyval 2 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI\_Type\_delete\_attr(datatype, type\_keyval, ierror) BIND(C) 5TYPE(MPI\_Datatype), INTENT(IN) :: datatype 6 INTEGER, INTENT(IN) :: type\_keyval 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI\_TYPE\_DUP\_FN(oldtype, type\_keyval, extra\_state, attribute\_val\_in, 10 attribute\_val\_out, flag, ierror) BIND(C) 11 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype INTEGER, INTENT(IN) :: type\_keyval 12INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state, 13 14attribute\_val\_in 15INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val\_out 16LOGICAL, INTENT(OUT) :: flag 17INTEGER, INTENT(OUT) :: ierror 18 MPI\_Type\_free\_keyval(type\_keyval, ierror) BIND(C) 19 INTEGER, INTENT(INOUT) :: type\_keyval 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI\_Type\_get\_attr(datatype, type\_keyval, attribute\_val, flag, ierror) 23BIND(C) 24TYPE(MPI\_Datatype), INTENT(IN) :: datatype 25INTEGER, INTENT(IN) :: type\_keyval 26INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val 27LOGICAL, INTENT(OUT) :: flag 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 MPI\_Type\_get\_name(datatype, type\_name, resultlen, ierror) BIND(C) 30 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 31CHARACTER(LEN=MPI\_MAX\_OBJECT\_NAME), INTENT(OUT) :: type\_name 32 INTEGER, INTENT(OUT) :: resultlen 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI\_TYPE\_NULL\_COPY\_FN(oldtype, type\_keyval, extra\_state, attribute\_val\_in, 36 attribute\_val\_out, flag, ierror) BIND(C) 37 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 38 INTEGER, INTENT(IN) :: type\_keyval 39 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state, 40attribute\_val\_in 41 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val\_out 42LOGICAL, INTENT(OUT) :: flag 43 INTEGER, INTENT(OUT) :: ierror 44 MPI\_TYPE\_NULL\_DELETE\_FN(datatype, type\_keyval, attribute\_val, extra\_state, 45ierror) BIND(C) 46TYPE(MPI\_Datatype), INTENT(IN) :: datatype 47INTEGER, INTENT(IN) :: type\_keyval 48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
2
         extra_state
3
         INTEGER, INTENT(OUT) :: ierror
4
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         INTEGER, INTENT(IN) :: type_keyval
7
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
    MPI_Type_set_name(datatype, type_name, ierror) BIND(C)
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         CHARACTER(LEN=*), INTENT(IN) :: type_name
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
15
                  extra_state, ierror) BIND(C)
16
         PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
17
         PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
18
         INTEGER, INTENT(OUT) :: win_keyval
19
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
    MPI_Win_delete_attr(win, win_keyval, ierror) BIND(C)
23
         TYPE(MPI_Win), INTENT(IN) :: win
24
         INTEGER, INTENT(IN) :: win_keyval
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
    MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
27
                  attribute_val_out, flag, ierror) BIND(C)
28
         INTEGER, INTENT(IN) :: oldwin, win_keyval
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
30
         attribute_val_in
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
32
         LOGICAL, INTENT(OUT) :: flag
33
         INTEGER, INTENT(OUT) :: ierror
34
35
     MPI_Win_free_keyval(win_keyval, ierror) BIND(C)
36
         INTEGER, INTENT(INOUT) :: win_keyval
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
    MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror) BIND(C)
39
         TYPE(MPI_Win), INTENT(IN) :: win
40
         INTEGER, INTENT(IN) :: win_keyval
41
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
42
         LOGICAL, INTENT(OUT) :: flag
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Win_get_name(win, win_name, resultlen, ierror) BIND(C)
46
         TYPE(MPI_Win), INTENT(IN) :: win
47
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
48
```

```
1
    INTEGER, INTENT(OUT) :: resultlen
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror) BIND(C)
                                                                                 5
    INTEGER, INTENT(IN) :: oldwin, win_keyval
                                                                                 6
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
                                                                                 7
    attribute_val_in
                                                                                 8
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
                                                                                 9
    LOGICAL, INTENT(OUT) :: flag
                                                                                 10
    INTEGER, INTENT(OUT) :: ierror
                                                                                 11
MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
                                                                                 12
                                                                                 13
             BIND(C)
                                                                                 14
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 15
    INTEGER, INTENT(IN) :: win_keyval
                                                                                 16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
                                                                                 17
    extra_state
                                                                                 18
    INTEGER, 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
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
                                                                                 23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 24
                                                                                 25
MPI_Win_set_name(win, win_name, ierror) BIND(C)
                                                                                 26
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 27
    CHARACTER(LEN=*), INTENT(IN) :: win_name
                                                                                 28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 29
                                                                                 30
A.3.5 Process Topologies Fortran 2008 Bindings
                                                                                 31
                                                                                 32
MPI_Cart_coords(comm, rank, maxdims, coords, ierror) BIND(C)
                                                                                 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
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                 40
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
                                                                                 41
    LOGICAL, INTENT(IN) :: periods(ndims), reorder
                                                                                 42
    TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
                                                                                 45
MPI_Cartdim_get(comm, ndims, ierror) BIND(C)
                                                                                 46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 47
    INTEGER, INTENT(OUT) :: ndims
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) BIND(C)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, INTENT(IN) :: maxdims
5
         INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
6
         LOGICAL, INTENT(OUT) :: periods(maxdims)
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) BIND(C)
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
12
         LOGICAL, INTENT(IN) :: periods(ndims)
13
         INTEGER, INTENT(OUT) :: newrank
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_Cart_rank(comm, coords, rank, ierror) BIND(C)
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, INTENT(IN) :: coords(*)
18
         INTEGER, INTENT(OUT) :: rank
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
22
                  BIND(C)
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         INTEGER, INTENT(IN) :: direction, disp
25
         INTEGER, INTENT(OUT) :: rank_source, rank_dest
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Cart_sub(comm, remain_dims, newcomm, ierror) BIND(C)
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         LOGICAL, INTENT(IN) :: remain_dims(*)
30
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
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_adjacent(comm_old, indegree, sources, sourceweights,
38
                  outdegree, destinations, destweights, info, reorder,
39
                  comm_dist_graph, ierror) BIND(C)
40
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
41
         INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
42
         destinations(outdegree)
43
         INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
44
         TYPE(MPI_Info), INTENT(IN) :: info
45
         LOGICAL, INTENT(IN) :: reorder
46
         TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
                                                                                 2
             info, reorder, comm_dist_graph, ierror) BIND(C)
                                                                                 3
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                 4
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
    INTEGER, INTENT(IN) :: weights(*)
                                                                                 5
                                                                                 6
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 7
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                 8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 9
                                                                                 10
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
                                                                                 11
             maxoutdegree, destinations, destweights, ierror) BIND(C)
                                                                                 12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 13
    INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
                                                                                 14
    INTEGER, INTENT(OUT) :: sources(maxindegree),
                                                                                 15
    destinations(maxoutdegree)
                                                                                 16
    INTEGER :: sourceweights(*), destweights(*)
                                                                                 17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 18
                                                                                 19
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
                                                                                 20
             BIND(C)
                                                                                 21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 22
    INTEGER, INTENT(OUT) :: indegree, outdegree
                                                                                 23
    LOGICAL, INTENT(OUT) :: weighted
                                                                                 24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 25
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
                                                                                 26
             ierror) BIND(C)
                                                                                 27
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                 28
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                 29
    LOGICAL, INTENT(IN) :: reorder
                                                                                 30
    TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
                                                                                 31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 32
                                                                                 33
MPI_Graphdims_get(comm, nnodes, nedges, ierror) BIND(C)
                                                                                 34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 35
    INTEGER, INTENT(OUT) :: nnodes, nedges
                                                                                 36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 37
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) BIND(C)
                                                                                 38
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 39
    INTEGER, INTENT(IN) :: maxindex, maxedges
                                                                                 40
    INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
                                                                                 43
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) BIND(C)
                                                                                 44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 45
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                 46
    INTEGER, INTENT(OUT) :: newrank
                                                                                 47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 48
```

```
1
    MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) BIND(C)
\mathbf{2}
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         INTEGER, INTENT(IN) :: rank, maxneighbors
4
         INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
    MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) BIND(C)
7
         TYPE(MPI_Comm), INTENT(IN) :: comm
8
         INTEGER, INTENT(IN) :: rank
9
         INTEGER, INTENT(OUT) :: nneighbors
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
13
                  recvtype, comm, request, ierror) BIND(C)
14
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
16
         INTEGER, INTENT(IN) :: sendcount, recvcount
17
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         TYPE(MPI_Request), INTENT(OUT) :: request
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
22
                  displs, recvtype, comm, request, ierror) BIND(C)
23
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
         INTEGER, INTENT(IN) :: sendcount
26
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
27
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
33
                  recvtype, comm, request, ierror) BIND(C)
34
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
36
         INTEGER, INTENT(IN) :: sendcount, recvcount
37
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Request), INTENT(OUT) :: request
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
42
                  recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)
43
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
44
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
45
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
46
         recvcounts(*), rdispls(*)
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 2
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                 5
             recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
                                                                                 6
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 7
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                 8
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                 9
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                 10
    sdispls(*), rdispls(*)
                                                                                 11
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                 12
    recvtypes(*)
                                                                                 13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 14
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 16
                                                                                 17
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                 18
             recvtype, comm, ierror) BIND(C)
                                                                                 19
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 20
                                                                                 21
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 22
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 25
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                 26
             displs, recvtype, comm, ierror) BIND(C)
                                                                                 27
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 28
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 29
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                 30
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 33
                                                                                 34
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                 35
             recvtype, comm, ierror) BIND(C)
                                                                                 36
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 37
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 38
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 39
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                 43
             recvcounts, rdispls, recvtype, comm, ierror) BIND(C)
                                                                                 44
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 45
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 46
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                 47
    rdispls(*)
                                                                                 48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
5
                  recvcounts, rdispls, recvtypes, comm, ierror) BIND(C)
6
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
7
         TYPE(*), DIMENSION(..) :: recvbuf
8
         INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Topo_test(comm, status, ierror) BIND(C)
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, INTENT(OUT) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     A.3.6 MPI Environmental Management Fortran 2008 Bindings
20
21
    DOUBLE PRECISION MPI_Wtick() BIND(C)
22
    DOUBLE PRECISION MPI_Wtime() BIND(C)
23
^{24}
    MPI_Abort(comm, errorcode, ierror) BIND(C)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, INTENT(IN) :: errorcode
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Add_error_class(errorclass, ierror) BIND(C)
29
         INTEGER, INTENT(OUT) :: errorclass
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     MPI_Add_error_code(errorclass, errorcode, ierror) BIND(C)
33
         INTEGER, INTENT(IN) :: errorclass
34
         INTEGER, INTENT(OUT) :: errorcode
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
    MPI_Add_error_string(errorcode, string, ierror) BIND(C)
38
         INTEGER, INTENT(IN) :: errorcode
         CHARACTER(LEN=*), INTENT(IN) :: string
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
    MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
42
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
44
         TYPE(MPI_Info), INTENT(IN) :: info
45
         TYPE(C_PTR), INTENT(OUT) :: baseptr
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
     MPI_Comm_call_errhandler(comm, errorcode, ierror) BIND(C)
```

TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2 3
<pre>MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) BIND(C)     PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	4 5 6 7 8
<pre>MPI_Comm_get_errhandler(comm, errhandler, ierror) BIND(C)     TYPE(MPI_Comm), INTENT(IN) :: comm     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	9 10 11 12 13
<pre>MPI_Comm_set_errhandler(comm, errhandler, ierror) BIND(C)     TYPE(MPI_Comm), INTENT(IN) :: comm     TYPE(MPI_Errhandler), INTENT(IN) :: errhandler     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	14 15 16 17
<pre>MPI_Errhandler_free(errhandler, ierror) BIND(C)     TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	18 19 20 21
<pre>MPI_Error_class(errorcode, errorclass, ierror) BIND(C) INTEGER, INTENT(IN) :: errorcode INTEGER, INTENT(OUT) :: errorclass INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	22 23 24 25
<pre>MPI_Error_string(errorcode, string, resultlen, ierror) BIND(C) INTEGER, INTENT(IN) :: errorcode CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	26 27 28 29 30 31
<pre>MPI_File_call_errhandler(fh, errorcode, ierror) BIND(C)    TYPE(MPI_File), INTENT(IN) :: fh    INTEGER, INTENT(IN) :: errorcode    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	32 33 34 35
<pre>MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C)     PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	36 37 38 39 40
<pre>MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)    TYPE(MPI_File), INTENT(IN) :: file    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	41 42 43 44
<pre>MPI_File_set_errhandler(file, errhandler, ierror) BIND(C)     TYPE(MPI_File), INTENT(IN) :: file     TYPE(MPI_Errhandler), INTENT(IN) :: errhandler</pre>	45 46 47 48

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror  $\mathbf{2}$ MPI\_Finalized(flag, ierror) BIND(C) 3 LOGICAL, INTENT(OUT) :: flag 4 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 56 MPI\_Finalize(ierror) BIND(C) 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 MPI\_Free\_mem(base, ierror) BIND(C) 9 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 12MPI\_Get\_library\_version(version, resulten, ierror) BIND(C) 13CHARACTER(LEN=MPI\_MAX\_LIBRARY\_VERSION\_STRING), INTENT(OUT) :: version 14INTEGER, INTENT(OUT) :: resultlen 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI\_Get\_processor\_name(name, resultlen, ierror) BIND(C) 17CHARACTER(LEN=MPI\_MAX\_PROCESSOR\_NAME), INTENT(OUT) :: name 18 INTEGER, INTENT(OUT) :: resultlen 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_Get\_version(version, subversion, ierror) BIND(C) 22INTEGER, INTENT(OUT) :: version, subversion 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI\_Initialized(flag, ierror) BIND(C) 25LOGICAL, INTENT(OUT) :: flag 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728 MPI\_Init(ierror) BIND(C) 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI\_Win\_call\_errhandler(win, errorcode, ierror) BIND(C)  $^{31}$ TYPE(MPI\_Win), INTENT(IN) :: win 32 INTEGER, INTENT(IN) :: errorcode 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI\_Win\_create\_errhandler(win\_errhandler\_fn, errhandler, ierror) BIND(C) 36 PROCEDURE(MPI\_Win\_errhandler\_function) :: win\_errhandler\_fn 37 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI\_Win\_get\_errhandler(win, errhandler, ierror) BIND(C) 40TYPE(MPI\_Win), INTENT(IN) :: win 41 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44MPI\_Win\_set\_errhandler(win, errhandler, ierror) BIND(C) 45TYPE(MPI\_Win), INTENT(IN) :: win 46TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

A.3.7 The Info Object Fortran 2008 Bindings	1
MPI_Info_create(info, ierror) BIND(C)	2
TYPE(MPI_Info), INTENT(OUT) :: info	3 4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4 5
MDI Info doloto (info low ionnon) DIND(C)	6
<pre>MPI_Info_delete(info, key, ierror) BIND(C)     TYPE(MPI_Info), INTENT(IN) :: info</pre>	7
CHARACTER(LEN=*), INTENT(IN) :: key	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
	10
MPI_Info_dup(info, newinfo, ierror) BIND(C)	11
TYPE(MPI_Info), INTENT(IN) :: info	12
TYPE(MPI_Info), INTENT(OUT) :: newinfo INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
INTEGER, OPTIONAL, INTENI(UOI) :: Terror	14
MPI_Info_free(info, ierror) BIND(C)	15
TYPE(MPI_Info), INTENT(INOUT) :: info	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17 18
MPI_Info_get(info, key, valuelen, value, flag, ierror) BIND(C)	19
TYPE(MPI_Info), INTENT(IN) :: info	20
CHARACTER(LEN=*), INTENT(IN) :: key	21
INTEGER, INTENT(IN) :: valuelen	22
CHARACTER(LEN=valuelen), INTENT(OUT) :: value	23
LOGICAL, INTENT(OUT) :: flag	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MPI_Info_get_nkeys(info, nkeys, ierror) BIND(C)	26
TYPE(MPI_Info), INTENT(IN) :: info	27
INTEGER, INTENT(OUT) :: nkeys	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
MPI_Info_get_nthkey(info, n, key, ierror) BIND(C)	30
TYPE(MPI_Info), INTENT(IN) :: info	31 32
INTEGER, INTENT(IN) :: n	33
CHARACTER(LEN=*), INTENT(OUT) :: key	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
MDI Info get upluglon (info key upluglon flog ignor) DIND(C)	36
<pre>MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) BIND(C)         TYPE(MPI_Info), INTENT(IN) :: info</pre>	37
CHARACTER(LEN=*), INTENT(IN) :: key	38
INTEGER, INTENT(OUT) :: valuelen	39
LOGICAL, INTENT(OUT) :: flag	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
	42
MPI_Info_set(info, key, value, ierror) BIND(C)	43
TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=*) INTENT(IN) :: kow walno	44
CHARACTER(LEN=*), INTENT(IN) :: key, value INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
INTEGER, OF ITOWRE, INTENT(UOT) LEITOT	46 47
	48

```
1
     A.3.8 Process Creation and Management Fortran 2008 Bindings
\mathbf{2}
     MPI_Close_port(port_name, ierror) BIND(C)
3
         CHARACTER(LEN=*), INTENT(IN) :: port_name
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) BIND(C)
7
         CHARACTER(LEN=*), INTENT(IN) :: port_name
8
         TYPE(MPI_Info), INTENT(IN) :: info
9
         INTEGER, INTENT(IN) :: root
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror) BIND(C)
14
         CHARACTER(LEN=*), INTENT(IN) :: port_name
15
         TYPE(MPI_Info), INTENT(IN) :: info
16
         INTEGER, INTENT(IN) :: root
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Comm_disconnect(comm, ierror) BIND(C)
22
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Comm_get_parent(parent, ierror) BIND(C)
25
         TYPE(MPI_Comm), INTENT(OUT) :: parent
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Comm_join(fd, intercomm, ierror) BIND(C)
29
         INTEGER, INTENT(IN) :: fd
30
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
33
                  array_of_errcodes, ierror) BIND(C)
34
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
35
         INTEGER, INTENT(IN) :: maxprocs, root
36
         TYPE(MPI_Info), INTENT(IN) :: info
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
39
         INTEGER :: array_of_errcodes(*)
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
43
                  array_of_maxprocs, array_of_info, root, comm, intercomm,
44
                  array_of_errcodes, ierror) BIND(C)
45
         INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
46
         CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
47
         array_of_argv(count, *)
48
```

```
1
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
                                                                                 2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 3
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER :: array_of_errcodes(*)
                                                                                 4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 5
                                                                                 6
MPI_Lookup_name(service_name, info, port_name, ierror) BIND(C)
                                                                                 7
    CHARACTER(LEN=*), INTENT(IN) :: service_name
                                                                                 8
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 9
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                11
MPI_Open_port(info, port_name, ierror) BIND(C)
                                                                                12
                                                                                13
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                14
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                16
MPI_Publish_name(service_name, info, port_name, ierror) BIND(C)
                                                                                17
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                18
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
                                                                                19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                20
                                                                                21
MPI_Unpublish_name(service_name, info, port_name, ierror) BIND(C)
                                                                                22
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
                                                                                23
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                25
                                                                                26
A.3.9 One-Sided Communications Fortran 2008 Bindings
                                                                                27
                                                                                28
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                29
             target_disp, target_count, target_datatype, op, win, ierror)
                                                                                30
             BIND(C)
                                                                                31
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                32
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                33
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                34
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                35
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                36
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                38
MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
                                                                                39
             target_rank, target_disp, win, ierror) BIND(C)
                                                                                40
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
                                                                                41
    compare_addr
                                                                                42
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                43
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                44
    INTEGER, INTENT(IN) :: target_rank
                                                                                45
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                46
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                48
```

```
1
    MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
\mathbf{2}
                  target_disp, op, win, ierror) BIND(C)
3
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
4
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         INTEGER, INTENT(IN) :: target_rank
7
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
8
         TYPE(MPI_Op), INTENT(IN) :: op
9
         TYPE(MPI_Win), INTENT(IN) :: win
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
    MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
12
                  result_count, result_datatype, target_rank, target_disp,
13
                  target_count, target_datatype, op, win, ierror) BIND(C)
14
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
16
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
17
         target_count
18
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
19
         result_datatype
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
21
         TYPE(MPI_Op), INTENT(IN) :: op
22
         TYPE(MPI_Win), INTENT(IN) :: win
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
    MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
26
                  target_disp, target_count, target_datatype, win, ierror)
27
                  BIND(C)
28
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
29
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
30
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
32
         TYPE(MPI_Win), INTENT(IN) :: win
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
    MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
35
                  target_disp, target_count, target_datatype, win, ierror)
36
                  BIND(C)
37
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
38
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
39
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
40
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
41
         TYPE(MPI_Win), INTENT(IN) :: win
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
    MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
45
                  target_disp, target_count, target_datatype, op, win, request,
46
                  ierror) BIND(C)
47
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
48
```

1 INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 2 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Op), INTENT(IN) :: op 4 TYPE(MPI\_Win), INTENT(IN) :: win 56 TYPE(MPI\_Request), INTENT(OUT) :: request 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 9 result\_addr, result\_count, result\_datatype, target\_rank, 10 target\_disp, target\_count, target\_datatype, op, win, request, 11 ierror) BIND(C) 12TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 13 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 14INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 15target\_count 16TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype, 17result\_datatype 18 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 19 TYPE(MPI\_Op), INTENT(IN) :: op 20TYPE(MPI\_Win), INTENT(IN) :: win 21TYPE(MPI\_Request), INTENT(OUT) :: request 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2324MPI\_Rget(origin\_addr, origin\_count, origin\_datatype, target\_rank, 25target\_disp, target\_count, target\_datatype, win, request, 26ierror) BIND(C) 27TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: origin\_addr 28 INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 29TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 30 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 31TYPE(MPI\_Win), INTENT(IN) :: win 32 TYPE(MPI\_Request), INTENT(OUT) :: request 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI\_Rput(origin\_addr, origin\_count, origin\_datatype, target\_rank, 35 target\_disp, target\_count, target\_datatype, win, request, 36 ierror) BIND(C) 37 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 38 INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 39 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 40 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 41 TYPE(MPI\_Win), INTENT(IN) :: win 42TYPE(MPI\_Request), INTENT(OUT) :: request 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4445MPI\_Win\_allocate\_shared(size, disp\_unit, info, comm, baseptr, win, ierror) 46BIND(C) 47USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
2
         INTEGER, INTENT(IN) :: disp_unit
3
         TYPE(MPI_Info), INTENT(IN) :: info
4
         TYPE(MPI_Comm), INTENT(IN) ::
                                        comm
5
         TYPE(C_PTR), INTENT(OUT) :: baseptr
6
         TYPE(MPI_Win), INTENT(OUT) :: win
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
    MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) BIND(C)
9
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
11
         INTEGER, INTENT(IN) :: disp_unit
12
         TYPE(MPI_Info), INTENT(IN) :: info
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(C_PTR), INTENT(OUT) :: baseptr
15
         TYPE(MPI_Win), INTENT(OUT) :: win
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
    MPI_Win_attach(win, base, size, ierror) BIND(C)
19
         TYPE(MPI_Win), INTENT(IN) :: win
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                         size
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
    MPI_Win_complete(win, ierror) BIND(C)
24
         TYPE(MPI_Win), INTENT(IN) :: win
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) BIND(C)
28
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
30
         INTEGER, INTENT(IN) :: disp_unit
31
         TYPE(MPI_Info), INTENT(IN) :: info
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Win), INTENT(OUT) :: win
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
    MPI_Win_create_dynamic(info, comm, win, ierror) BIND(C)
36
         TYPE(MPI_Info), INTENT(IN) :: info
37
         TYPE(MPI_Comm), INTENT(IN) ::
                                        comm
38
         TYPE(MPI_Win), INTENT(OUT) :: win
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Win_detach(win, base, ierror) BIND(C)
42
         TYPE(MPI_Win), INTENT(IN) :: win
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_Win_fence(assert, win, ierror) BIND(C)
46
         INTEGER, INTENT(IN) :: assert
47
         TYPE(MPI_Win), INTENT(IN) :: win
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Win_flush_all(win, ierror) BIND(C)	2 3
TYPE(MPI_Win), INTENT(IN) :: win	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
MDT the fluck least all (win issues) DIND(C)	6
<pre>MPI_Win_flush_local_all(win, ierror) BIND(C)     TYPE(MPI_Win), INTENT(IN) :: win</pre>	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
	9
MPI_Win_flush_local(rank, win, ierror) BIND(C)	10
INTEGER, INTENT(IN) :: rank	11
TYPE(MPI_Win), INTENT(IN) :: win	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Win_flush(rank, win, ierror) BIND(C)	14
INTEGER, INTENT(IN) :: rank	15
TYPE(MPI_Win), INTENT(IN) :: win	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_Win_free(win, ierror) BIND(C)	18 19
TYPE(MPI_Win), INTENT(INOUT) :: win	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MDT Uin not moun (min moun ichnow) DIND(C)	22
<pre>MPI_Win_get_group(win, group, ierror) BIND(C)     TYPE(MPI_Win), INTENT(IN) :: win</pre>	23
TYPE(MPI_WIN), INTENT(UN) win TYPE(MPI_Group), INTENT(OUT) :: group	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
	26
MPI_Win_lock_all(assert, win, ierror) BIND(C)	27
INTEGER, INTENT(IN) :: assert	28
TYPE(MPI_Win), INTENT(IN) :: win	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
<pre>MPI_Win_lock(lock_type, rank, assert, win, ierror) BIND(C)</pre>	31 32
<pre>INTEGER, INTENT(IN) :: lock_type, rank, assert</pre>	32
TYPE(MPI_Win), INTENT(IN) :: win	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
MPI_Win_post(group, assert, win, ierror) BIND(C)	36
TYPE(MPI_Group), INTENT(IN) :: group	37
INTEGER, INTENT(IN) :: assert	38
TYPE(MPI_Win), INTENT(IN) :: win	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) BIND(C)	41
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	42
TYPE(MPI_Win), INTENT(IN) :: win	43
INTEGER, INTENT(IN) :: rank	44
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size	45 46
INTEGER, INTENT(OUT) :: disp_unit	40 47
TYPE(C_PTR), INTENT(OUT) :: baseptr	48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Win_start(group, assert, win, ierror) BIND(C)
3
         TYPE(MPI_Group), INTENT(IN) :: group
4
         INTEGER, INTENT(IN) :: assert
5
         TYPE(MPI_Win), INTENT(IN) :: win
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Win_sync(win, ierror) BIND(C)
9
         TYPE(MPI_Win), INTENT(IN) :: win
10
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
11
     MPI_Win_test(win, flag, ierror) BIND(C)
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         LOGICAL, INTENT(OUT) :: flag
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
    MPI_Win_unlock_all(win, ierror) BIND(C)
17
         TYPE(MPI_Win), INTENT(IN) :: win
18
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
19
    MPI_Win_unlock(rank, win, ierror) BIND(C)
20
         INTEGER, INTENT(IN) :: rank
21
         TYPE(MPI_Win), INTENT(IN) :: win
22
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
23
24
     MPI_Win_wait(win, ierror) BIND(C)
25
         TYPE(MPI_Win), INTENT(IN) :: win
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     A.3.10 External Interfaces Fortran 2008 Bindings
29
30
    MPI_Grequest_complete(request, ierror) BIND(C)
31
         TYPE(MPI_Request), INTENT(IN) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
                   ierror) BIND(C)
35
36
         PROCEDURE(MPI_Grequest_query_function) :: query_fn
37
         PROCEDURE(MPI_Grequest_free_function) :: free_fn
38
         PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
39
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Init_thread(required, provided, ierror) BIND(C)
43
         INTEGER, INTENT(IN) :: required
44
         INTEGER, INTENT(OUT) :: provided
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
    MPI_Is_thread_main(flag, ierror) BIND(C)
48
         LOGICAL, INTENT(OUT) :: flag
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Query_thread(provided, ierror) BIND(C)	2
INTEGER, INTENT(OUT) :: provided	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
	5 6
MPI_Status_set_cancelled(status, flag, ierror) BIND(C)	7
TYPE(MPI_Status), INTENT(INOUT) :: status	8
LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
INTEGER, OFFICIARE, INTENI(001) TEITOI	10
<pre>MPI_Status_set_elements(status, datatype, count, ierror) BIND(C)</pre>	11
TYPE(MPI_Status), INTENT(INOUT) :: status	12
TYPE(MPI_Datatype), INTENT(IN) :: datatype	13
INTEGER, INTENT(IN) :: count	14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15
MPI_Status_set_elements_x(status, datatype, count, ierror) BIND(C)	16
TYPE(MPI_Status), INTENT(INOUT) :: status	17
TYPE(MPI_Datatype), INTENT(IN) :: datatype	18
INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) :: count	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	21 22
A.3.11 I/O Fortran 2008 Bindings	22
	24
MPI_File_close(fh, ierror) BIND(C)	25
TYPE(MPI_File), INTENT(INOUT) :: fh	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
MPI_File_delete(filename, info, ierror) BIND(C)	28
CHARACTER(LEN=*), INTENT(IN) :: filename	29
TYPE(MPI_Info), INTENT(IN) :: info	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
MPI_File_get_amode(fh, amode, ierror) BIND(C)	32
TYPE(MPI_File), INTENT(IN) :: fh	33
INTEGER, INTENT(OUT) :: amode	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35 36
MDI File get stemicity(fb flog ierror) PIND(C)	30
<pre>MPI_File_get_atomicity(fh, flag, ierror) BIND(C)     TYPE(MPI_File), INTENT(IN) :: fh</pre>	38
LOGICAL, INTENT(OUT) :: flag	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	41
MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C)	42
TYPE(MPI_File), INTENT(IN) :: fh	43
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	44
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
INITORY, OLITOWER, INTENI(OOI) TELLOI	46
MPI_File_get_group(fh, group, ierror) BIND(C)	47
TYPE(MPI_File), INTENT(IN) :: fh	48

```
1
         TYPE(MPI_Group), INTENT(OUT) :: group
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_get_info(fh, info_used, ierror) BIND(C)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         TYPE(MPI_Info), INTENT(OUT) :: info_used
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_File_get_position(fh, offset, ierror) BIND(C)
9
         TYPE(MPI_File), INTENT(IN) :: fh
10
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_File_get_position_shared(fh, offset, ierror) BIND(C)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_File_get_size(fh, size, ierror) BIND(C)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_File_get_type_extent(fh, datatype, extent, ierror) BIND(C)
22
         TYPE(MPI_File), INTENT(IN) :: fh
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) BIND(C)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
30
         TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
31
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
34
                  BIND(C)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                        offset
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
38
         INTEGER, INTENT(IN) :: count
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_File_iread(fh, buf, count, datatype, request, ierror) BIND(C)
44
         TYPE(MPI_File), INTENT(IN) :: fh
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
46
         INTEGER, INTENT(IN) :: count
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 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_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                 12
             BIND(C)
                                                                                 13
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 14
                                                                                 15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 16
    INTEGER, INTENT(IN) :: count
                                                                                 17
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 20
MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)
                                                                                 21
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 22
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 23
    INTEGER, INTENT(IN) :: count
                                                                                 24
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 27
                                                                                 28
MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
                                                                                 29
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 30
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 31
    INTEGER, INTENT(IN) :: count
                                                                                 32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 35
MPI_File_open(comm, filename, amode, info, fh, ierror) BIND(C)
                                                                                 36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 37
    CHARACTER(LEN=*), INTENT(IN) :: filename
                                                                                 38
    INTEGER, INTENT(IN) :: amode
                                                                                 39
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 40
    TYPE(MPI_File), INTENT(OUT) :: fh
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
                                                                                 43
MPI_File_preallocate(fh, size, ierror) BIND(C)
                                                                                 44
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 45
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
                                                                                 46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 47
MPI_File_read_all_begin(fh, buf, count, datatype, ierror) BIND(C)
                                                                                 48
```

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
3
         INTEGER, INTENT(IN) :: count
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
    MPI_File_read_all_end(fh, buf, status, ierror) BIND(C)
7
         TYPE(MPI_File), INTENT(IN) :: fh
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
9
         TYPE(MPI_Status) :: status
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_File_read_all(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_at_all_begin(fh, offset, buf, count, datatype, ierror)
20
                  BIND(C)
21
         TYPE(MPI_File), INTENT(IN) :: fh
22
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
24
         INTEGER, INTENT(IN) :: count
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_File_read_at_all_end(fh, buf, status, ierror) BIND(C)
29
         TYPE(MPI_File), INTENT(IN) :: fh
30
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
31
         TYPE(MPI_Status) :: status
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
34
                  BIND(C)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
37
         TYPE(*), DIMENSION(..) :: buf
38
         INTEGER, INTENT(IN) :: count
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Status) :: status
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
    MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
44
         TYPE(MPI_File), INTENT(IN) :: fh
45
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
46
         TYPE(*), DIMENSION(..) :: buf
47
         INTEGER, INTENT(IN) :: count
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 2
    TYPE(MPI_Status) :: status
                                                                                 3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
MPI_File_read(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                 5
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 6
    TYPE(*), DIMENSION(..) :: buf
                                                                                 7
    INTEGER, INTENT(IN) :: count
                                                                                 8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 9
    TYPE(MPI_Status) :: status
                                                                                10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                11
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
                                                                                12
                                                                                13
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                15
    INTEGER, INTENT(IN) :: count
                                                                                16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                18
MPI_File_read_ordered_end(fh, buf, status, ierror) BIND(C)
                                                                                19
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                20
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                21
    TYPE(MPI_Status) :: status
                                                                                22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                23
                                                                                24
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                25
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                26
    TYPE(*), DIMENSION(..) :: buf
                                                                                27
    INTEGER, INTENT(IN) :: count
                                                                                28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                29
    TYPE(MPI_Status) :: status
                                                                                30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                31
MPI_File_read_shared(fh, buf, count, datatype, status, ierror) BIND(C)
                                                                                32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                33
    TYPE(*), DIMENSION(..) :: buf
                                                                                34
    INTEGER, INTENT(IN) :: count
                                                                                35
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                36
    TYPE(MPI_Status) :: status
                                                                                37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                38
                                                                                39
MPI_File_seek(fh, offset, whence, ierror) BIND(C)
                                                                                40
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                41
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                42
    INTEGER, INTENT(IN) :: whence
                                                                                43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                44
MPI_File_seek_shared(fh, offset, whence, ierror) BIND(C)
                                                                                45
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                46
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                47
    INTEGER, INTENT(IN) :: whence
                                                                                48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_File_set_atomicity(fh, flag, ierror) BIND(C)
3
         TYPE(MPI_File), INTENT(IN) :: fh
4
         LOGICAL, INTENT(IN) :: flag
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_File_set_info(fh, info, ierror) BIND(C)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
    MPI_File_set_size(fh, size, ierror) BIND(C)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
19
         TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
20
         CHARACTER(LEN=*), INTENT(IN) :: datarep
21
         TYPE(MPI_Info), INTENT(IN) :: info
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_File_sync(fh, ierror) BIND(C)
24
         TYPE(MPI_File), INTENT(IN) :: fh
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror) BIND(C)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
30
         INTEGER, INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)
34
         TYPE(MPI_File), INTENT(IN) :: fh
35
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
36
         TYPE(MPI_Status) :: status
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_File_write_all(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_at_all_begin(fh, offset, buf, count, datatype, ierror)
47
                  BIND(C)
48
```

1 TYPE(MPI\_File), INTENT(IN) :: fh 2 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 3 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count 4 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 5INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 7 MPI\_File\_write\_at\_all\_end(fh, buf, status, ierror) BIND(C) 8 TYPE(MPI\_File), INTENT(IN) :: fh 9 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 10TYPE(MPI\_Status) :: status 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI\_File\_write\_at\_all(fh, offset, buf, count, datatype, status, ierror) 14BIND(C) 15TYPE(MPI\_File), INTENT(IN) :: fh 16INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 17TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 18 INTEGER, INTENT(IN) :: count 19 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 20TYPE(MPI\_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI\_File\_write\_at(fh, offset, buf, count, datatype, status, ierror) BIND(C) 23TYPE(MPI\_File), INTENT(IN) :: fh 24INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 25TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 26INTEGER, INTENT(IN) :: count 27TYPE(MPI\_Datatype), INTENT(IN) :: datatype 28 TYPE(MPI\_Status) :: status 29INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31MPI\_File\_write(fh, buf, count, datatype, status, ierror) BIND(C) 32 TYPE(MPI\_File), INTENT(IN) :: fh 33 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 34 INTEGER, INTENT(IN) :: count 35 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI\_Status) :: status 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI\_File\_write\_ordered\_begin(fh, buf, count, datatype, ierror) BIND(C) 39 TYPE(MPI\_File), INTENT(IN) :: fh 40TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 41 INTEGER, INTENT(IN) :: count 42TYPE(MPI\_Datatype), INTENT(IN) :: datatype 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4445MPI\_File\_write\_ordered\_end(fh, buf, status, ierror) BIND(C) 46TYPE(MPI\_File), INTENT(IN) :: fh 47TYPE(\*), 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_shared(fh, buf, count, datatype, status, ierror) BIND(C)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
14
         INTEGER, INTENT(IN) :: count
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
19
                  dtype_file_extent_fn, extra_state, ierror) BIND(C)
20
         CHARACTER(LEN=*), INTENT(IN) :: datarep
21
         PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
22
         PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
23
         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
28
     A.3.12 Language Bindings Fortran 2008 Bindings
29
    MPI_F_sync_reg(buf) BIND(C)
30
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
^{31}
32
     MPI_Sizeof(x, size, ierror) BIND(C)
33
         TYPE(*), DIMENSION(..) :: x
34
         INTEGER, INTENT(OUT) :: size
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
    MPI_Status_f082f(f08_status, f_status, ierror) BIND(C)
37
         TYPE(MPI_Status), INTENT(IN) :: f08_status
38
         INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C)
42
         INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
43
         TYPE(MPI_Status), INTENT(OUT) :: f08_status
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
46
47
         INTEGER, INTENT(IN) :: p, r
48
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)	2 3
INTEGER, INTENT(IN) :: r	4
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C)	7
INTEGER, INTENT(IN) :: p, r	8
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10 11
MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C)	12
<pre>INTEGER, INTENT(IN) :: typeclass, size</pre>	13
TYPE(MPI_Datatype), INTENT(OUT) :: datatype	14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15
	16
A.3.13 Tools / Profiling Interface Fortran 2008 Bindings	17
MPI_Pcontrol(level) BIND(C)	18 19
INTEGER, INTENT(IN) :: level	20
	21
	22
	23
	24
	25
	26 27
	21
	29
	30
	31
	32
	33
	34 35
	36
	37
	38
	39
	40
	41
	42 43
	43 44
	45
	46
	47

Fortran Bindings with mpif.h or the mpi Module 1A.4  $\mathbf{2}$ 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 18 MPI\_CANCEL(REQUEST, IERROR) 19 INTEGER REQUEST, IERROR 20MPI\_GET\_COUNT(STATUS, DATATYPE, COUNT, IERROR) 2122INTEGER 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) 28INTEGER 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(\*) 43 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 44MPI\_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 45<type> BUF(\*) 46 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4748

**Unofficial Draft for Comment Only** 

## 758

<pre>MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
<pre>MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)</pre>
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
<pre>MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)</pre>
IERROR MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR</type>
MPI_REQUEST_FREE(REQUEST, IERROR) <sup>2</sup> INTEGER REQUEST, IERROR <sup>2</sup>
MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)       2         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR       2         LOGICAL FLAG       2
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <pre></pre>
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <pre>     type&gt; BUF(*)     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR     33 </pre>
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <pre></pre>
<pre>MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <pre></pre></pre>
<pre>MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,</type></pre>
STATUS(MPI_STATUS_SIZE), IERROR
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)

1<type> SENDBUF(\*), RECVBUF(\*)  $\mathbf{2}$ INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 3 SOURCE, RECVTAG, COMM, STATUS(MPI\_STATUS\_SIZE), 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\_STARTALL(COUNT, ARRAY\_OF\_REQUESTS, IERROR) 12INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), IERROR 13 14MPI\_START(REQUEST, IERROR) 15INTEGER REQUEST, IERROR 16MPI\_TESTALL(COUNT, ARRAY\_OF\_REQUESTS, FLAG, ARRAY\_OF\_STATUSES, IERROR) 17LOGICAL FLAG 18 INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), 19 ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE,\*), IERROR 2021MPI\_TESTANY(COUNT, ARRAY\_OF\_REQUESTS, INDEX, FLAG, STATUS, IERROR) 22LOGICAL FLAG 23INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), INDEX, STATUS(MPI\_STATUS\_SIZE),  $^{24}$ IERROR 25MPI\_TEST\_CANCELLED(STATUS, FLAG, IERROR) 26LOGICAL FLAG 27INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR 2829MPI\_TEST(REQUEST, FLAG, STATUS, IERROR) 30 LOGICAL FLAG 31INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR 32 MPI\_TESTSOME(INCOUNT, ARRAY\_OF\_REQUESTS, OUTCOUNT, ARRAY\_OF\_INDICES, 33 ARRAY\_OF\_STATUSES, IERROR) 34 INTEGER INCOUNT, ARRAY\_OF\_REQUESTS(\*), OUTCOUNT, ARRAY\_OF\_INDICES(\*), 35ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE,\*), IERROR 36 37 MPI\_WAITALL(COUNT, ARRAY\_OF\_REQUESTS, ARRAY\_OF\_STATUSES, IERROR) 38 INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*) 39 INTEGER ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE,\*), IERROR 40MPI\_WAITANY(COUNT, ARRAY\_OF\_REQUESTS, INDEX, STATUS, IERROR) 41 INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), INDEX, STATUS(MPI\_STATUS\_SIZE), 42IERROR 43 44MPI\_WAIT(REQUEST, STATUS, IERROR) 45INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR 4647MPI\_WAITSOME(INCOUNT, ARRAY\_OF\_REQUESTS, OUTCOUNT, ARRAY\_OF\_INDICES, 48ARRAY\_OF\_STATUSES, IERROR)

INTEGER INCOUNT, ARRAY\_OF\_REQUESTS(\*), OUTCOUNT, ARRAY\_OF\_INDICES(\*),
ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE,\*), IERROR

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
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)	10
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	11 12
	13
MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)	14
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT	15
	16
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	17
POSITION, IERROR)	18
INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION	19
CHARACTER*(*) DATAREP	20 21
<type> INBUF(*), OUTBUF(*)</type>	21
	23
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)	24
INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE	25
CHARACTER*(*) DATAREP	26
	27
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)	28
<type> INBUF(*), OUTBUF(*) INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR</type>	29 30
INTEGER INCOUNT, DATATIFE, OUTSIZE, FUSTITION, COPPT, TERROR	31
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)	32
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR	33
MPI_TYPE_COMMIT(DATATYPE, IERROR)	34
INTEGER DATATYPE, IERROR	35
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)	36
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR	37
	38 39
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, 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
	44
MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	45
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	46
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	47
	48

1

 $\frac{2}{3}$ 

1	MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
2	ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
3	INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
5 6 7	MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
8	INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
9	INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
10	MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
11	OLDTYPE, NEWTYPE, IERROR)
12	INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
13	NEWTYPE, IERROR
14	MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
15	INTEGER OLDTYPE, NEWTYPE, IERROR
16	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
17 18	
19	<pre>MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>
20	
21 22	<pre>IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)</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	MPI_TYPE_FREE(DATATYPE, IERROR)
32	INTEGER DATATYPE, IERROR
33	MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
34	ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
35	IERROR)
36	<pre>INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,</pre>
37	ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
38	INTEGER(KIND_MDL_ADDRESS KIND) ADDAY OF ADDRESSES(*)
39	<pre>INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)</pre>
40	MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
41	COMBINER, IERROR)
42 43	INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR
44	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
45	INTEGER DATATYPE, IERROR
46	INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
47 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	4 5
INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT	6
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)	7 8
INTEGER DATATYPE, IERROR 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
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),	12 13
OLDTYPE, NEWTYPE, IERROR	14
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)	15
INTEGER DATATYPE, SIZE, IERROR	16 17
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)	18
INTEGER DATATYPE, IERROR INTEGER(KIND = MPI_COUNT_KIND) SIZE	19
	20 21
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	21 22 23
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, IERROR)	23 24 25
INTEGER OUTCOUNT, DATATYPE, IERROR	25 26
INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION	27
CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*)</type>	28
	29 30
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR)	31
<pre>(type&gt; INBUF(*), OUTBUF(*)</pre>	32
INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR	33 34
	34 35
A.4.3 Collective Communication Fortran Bindings	36
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	37
COMM, IERROR)	38 39
<type> SENDBUF(*), RECVBUF(*)</type>	40
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	41
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	42 43
RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>	43 44
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	45
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
<pre><type> SENDBUF(*), RECVBUF(*) INTEGED GENERATION FROM RECVENT FRO</type></pre>	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
<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>	18
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),	19
RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR	20
MPI_IBARRIER(COMM, REQUEST, IERROR)	21
INTEGER COMM, REQUEST, IERROR	22
	23
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)	24
<type> BUFFER(*)</type>	25
INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR	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(*) </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_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,	42
REQUEST, IERROR)	43
<pre><type> SENDBUF(*), RECVBUF(*) INTECEP RECVCUINT DATATYDE OD COMM REQUEST IERROR</type></pre>	44
INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR	45
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,	46
REQUEST, IERROR)	47
	48

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
3
     MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
4
                   IERROR)
5
         <type> SENDBUF(*), RECVBUF(*)
6
         INTEGER COUNT, DATATYPE, OP, ROOT, 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_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
34
         <type> INBUF(*), INOUTBUF(*)
35
         INTEGER COUNT, DATATYPE, OP, IERROR
36
     MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
37
                   IERROR)
38
         <type> SENDBUF(*), RECVBUF(*)
39
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
40
41
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
42
                   IERROR)
43
         <type> SENDBUF(*), RECVBUF(*)
44
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
45
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
46
         <type> SENDBUF(*), RECVBUF(*)
47
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
48
```

<pre>MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)</pre>	1 2 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	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
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
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	2
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR	$\frac{3}{4}$
MDT COMM TECT INTED (COMM ELAC IEDDOD)	5
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR) 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)	13
INTEGER GROUP, N, RANKS(*), NEWGROUP, 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
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)	21 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 31
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)	32
INTEGER GROUP, SIZE, IERROR	33
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)	34
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR	35
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)	36
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	37
,	38

MPI\_INTERCOMM\_CREATE(LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER, TAG, NEWINTERCOMM, IERROR) INTEGER LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER, TAG, NEWINTERCOMM, IERROR

MPI\_INTERCOMM\_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH

MPI\_TYPE\_CREATE\_KEYVAL(TYPE\_COPY\_ATTR\_FN, TYPE\_DELETE\_ATTR\_FN, TYPE\_KEYVAL,

39 40

41 42

43

44

45

 $46 \\ 47$ 

1 2 3 4 5	EXTRA_STATE, IERROR) EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN INTEGER TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
6 7	MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR
8 9 10 11 12 13	<pre>MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>
14 15 16	MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR) INTEGER TYPE_KEYVAL, IERROR
17 18 19 20 21	MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL LOGICAL FLAG
22 22 23 24	MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR CHARACTER*(*) TYPE_NAME
25 26 27 28 29 30 31	<pre>MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>
32 33 34 35	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
36 37 38 39	MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
40 41 42	MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME
43 44 45 46 47 48	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 INTEGER WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

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
<pre>MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>	3 4 5 6 7 8 9
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR) INTEGER WIN_KEYVAL, IERROR	10 11 12
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
<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_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR) INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR	40 41 42
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERR INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR LOGICAL PERIODS(*), REORDER	OR) 43 44 45
MPI_CARTDIM_GET(COMM, NDIMS, IERROR) INTEGER COMM, NDIMS, IERROR	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 19 MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(COMM\_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, 20OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, 21COMM\_DIST\_GRAPH, IERROR) 22INTEGER COMM\_OLD, INDEGREE, SOURCES(\*), SOURCEWEIGHTS(\*), OUTDEGREE, 23DESTINATIONS(\*), DESTWEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR 24LOGICAL REORDER 25MPI\_DIST\_GRAPH\_CREATE(COMM\_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS, 26INFO, REORDER, COMM\_DIST\_GRAPH, IERROR) 27INTEGER COMM\_OLD, N, SOURCES(\*), DEGREES(\*), DESTINATIONS(\*), 28WEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR 29 LOGICAL 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 35MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) 36 INTEGER COMM, INDEGREE, OUTDEGREE, IERROR 37 LOGICAL WEIGHTED 3839 MPI\_GRAPH\_CREATE(COMM\_OLD, NNODES, INDEX, EDGES, REORDER, COMM\_GRAPH. 40IERROR) 41 INTEGER COMM\_OLD, NNODES, INDEX(\*), EDGES(\*), COMM\_GRAPH, IERROR 42LOGICAL REORDER 43 MPI\_GRAPHDIMS\_GET(COMM, NNODES, NEDGES, IERROR) 44INTEGER COMM, NNODES, NEDGES, IERROR 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 2 INTEGER COMM, NNODES, INDEX(\*), EDGES(\*), NEWRANK, IERROR 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 MPI\_INEIGHBOR\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 9 RECVTYPE, COMM, REQUEST, IERROR) 10 <type> 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<tvpe> 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 23 MPI\_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
^{24}
         INTEGER COMM, ERRORCODE, IERROR
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
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)	1
EXTERNAL COMM_ERRHANDLER_FN	2
INTEGER ERRHANDLER, IERROR	3
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)	4
INTEGER COMM, ERRHANDLER, IERROR	5
	6
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)	7
INTEGER COMM, ERRHANDLER, IERROR	8
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)	9
INTEGER ERRHANDLER, IERROR	10
	11
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)	12 13
INTEGER ERRORCODE, ERRORCLASS, IERROR	13
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)	14
INTEGER ERRORCODE, RESULTLEN, IERROR	16
CHARACTER*(*) STRING	17
	18
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)	19
INTEGER FH, ERRORCODE, IERROR	20
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)	21
EXTERNAL FILE_ERRHANDLER_FN	22
INTEGER ERRHANDLER, IERROR	23
	24
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	25
INTEGER FILE, ERRHANDLER, IERROR	26
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	27
INTEGER FILE, ERRHANDLER, IERROR	28
MPI_FINALIZED(FLAG, IERROR)	29
LOGICAL FLAG	30
INTEGER IERROR	31
	32
MPI_FINALIZE(IERROR)	33
INTEGER IERROR	34
MPI_FREE_MEM(BASE, IERROR)	35
<pre><type> BASE(*)</type></pre>	36
INTEGER IERROR	37
	38
MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR)	39
CHARACTER*(*) VERSION	40
INTEGER RESULTLEN, IERROR	41
MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)	42
CHARACTER*(*) NAME	43
INTEGER RESULTLEN, IERROR	44
	45
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)	46
INTEGER VERSION, SUBVERSION, IERROR	47
	48

```
1
     MPI_INITIALIZED(FLAG, IERROR)
\mathbf{2}
         LOGICAL FLAG
3
         INTEGER IERROR
4
     MPI_INIT(IERROR)
5
         INTEGER IERROR
6
\overline{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
27
     MPI_INFO_DUP(INFO, NEWINFO, IERROR)
         INTEGER INFO, NEWINFO, IERROR
28
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
A.4.8 Process Creation and Management Fortran Bindings	4
	6
MPI_CLOSE_PORT(PORT_NAME, IERROR) CHARACTER*(*) PORT_NAME	7
INTEGER IERROR	8
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)	9
CHARACTER*(*) PORT_NAME	10 11
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR	11
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)	13
CHARACTER*(*) PORT_NAME	14 15
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR	16
MPI_COMM_DISCONNECT(COMM, IERROR)	17
INTEGER COMM, IERROR	18
MPI_COMM_GET_PARENT(PARENT, IERROR)	19
INTEGER PARENT, IERROR	20 21
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)	21
INTEGER FD, INTERCOMM, IERROR	23
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,	24
ARRAY_OF_ERRCODES, IERROR)	25
CHARACTER*(*) COMMAND, ARGV(*)	26 27
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),	28
IERROR	29
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,	30
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,	31
ARRAY_OF_ERRCODES, IERROR) INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,	32
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
INTEGER INFO, IERROR	38 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)	43
INTEGER INFO, IERROR	44 45
CHARACTER*(*) SERVICE_NAME, PORT_NAME	40
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	47
INTEGER INFO, IERROR	48

```
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_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
23
                   RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
24
                   TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
25
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
26
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
27
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
28
         TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
29
30
     MPI_GET(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
    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
```

1 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 2 TARGET\_DATATYPE, OP, WIN, REQUEST, IERROR 3 MPI\_RGET\_ACCUMULATE(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, 4 RESULT\_ADDR, RESULT\_COUNT, RESULT\_DATATYPE, TARGET\_RANK, 5 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, 6 IERROR) 7 <type> ORIGIN\_ADDR(\*), RESULT\_ADDR(\*) 8 INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 9 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_COUNT, RESULT\_DATATYPE, 10 TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, IERROR 11 MPI\_RGET(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 1213 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, REQUEST, 14IERROR) 15<type> ORIGIN\_ADDR(\*) 16INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 17 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 18 TARGET\_DATATYPE, WIN, REQUEST, IERROR 19 MPI\_RPUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 20TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, REQUEST, 21IERROR) 22 <type> ORIGIN\_ADDR(\*) 23INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 24INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 25TARGET\_DATATYPE, WIN, REQUEST, IERROR 2627MPI\_WIN\_ALLOCATE\_SHARED(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 28 INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR 29INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR 30 If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 31INTERFACE MPI\_WIN\_ALLOCATE\_SHARED 32 SUBROUTINE MPI\_WIN\_ALLOCATE\_SHARED\_CPTR(SIZE, DISP\_UNIT, INFO, COMM, & 33 BASEPTR, 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 4041 MPI\_WIN\_ALLOCATE(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 42INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR 43 INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR 44If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 45INTERFACE MPI\_WIN\_ALLOCATE 46SUBROUTINE MPI\_WIN\_ALLOCATE\_CPTR(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, & 47 WIN, IERROR) 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_ALL(WIN, IERROR)
28
         INTEGER WIN, IERROR
29
30
     MPI_WIN_FLUSH_LOCAL_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(RANK, WIN, IERROR)
36
         INTEGER RANK, 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_ALL(ASSERT, WIN, IERROR)
46
         INTEGER ASSERT, WIN, IERROR
47
48
     MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
```

INTEGER LOCK\_TYPE, RANK, ASSERT, WIN, IERROR 1 2 MPI\_WIN\_POST(GROUP, ASSERT, WIN, IERROR) 3 INTEGER GROUP, ASSERT, WIN, IERROR 4 5MPI\_WIN\_SET\_INFO(WIN, INFO, IERROR) 6 INTEGER WIN, INFO, IERROR 7 MPI\_WIN\_SHARED\_QUERY(WIN, RANK, SIZE, DISP\_UNIT, BASEPTR, IERROR) 8 INTEGER WIN, RANK, DISP\_UNIT, IERROR 9 INTEGER (KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR 10 If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 11 INTERFACE MPI\_WIN\_SHARED\_QUERY 12SUBROUTINE MPI\_WIN\_SHARED\_QUERY\_CPTR(WIN, RANK, SIZE, DISP\_UNIT, & 13 BASEPTR, IERROR) 14USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 15INTEGER :: WIN, RANK, DISP\_UNIT, IERROR 1617INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE 18 TYPE(C\_PTR) :: BASEPTR END SUBROUTINE 19 END INTERFACE 2021MPI\_WIN\_START(GROUP, ASSERT, WIN, IERROR) 22 INTEGER GROUP, ASSERT, WIN, IERROR 2324MPI\_WIN\_SYNC(WIN, IERROR) 25INTEGER WIN, IERROR 26MPI\_WIN\_TEST(WIN, FLAG, IERROR) 27INTEGER WIN, IERROR 28 LOGICAL FLAG 29 30 MPI\_WIN\_UNLOCK\_ALL(WIN, IERROR) 31INTEGER WIN, IERROR 32 MPI\_WIN\_UNLOCK(RANK, WIN, IERROR) 33 INTEGER RANK, WIN, IERROR 34 35 MPI\_WIN\_WAIT(WIN, IERROR) 36 INTEGER WIN, IERROR 37 38 A.4.10 External Interfaces Fortran Bindings 39 40 MPI\_GREQUEST\_COMPLETE(REQUEST, IERROR) 41 INTEGER REQUEST, IERROR 42MPI\_GREQUEST\_START(QUERY\_FN, FREE\_FN, CANCEL\_FN, EXTRA\_STATE, REQUEST, 43 44IERROR) 45INTEGER REQUEST, IERROR 46EXTERNAL QUERY\_FN, FREE\_FN, CANCEL\_FN 47INTEGER (KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE 48

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 9	MPI_QUERY_THREAD(PROVIDED, IERROR) INTEGER PROVIDED, IERROR
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

1 INTEGER(KIND=MPI\_OFFSET\_KIND) OFFSET 2 MPI\_FILE\_GET\_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR 4 INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE 5 6 MPI\_FILE\_GET\_TYPE\_EXTENT(FH, DATATYPE, EXTENT, IERROR) INTEGER FH, DATATYPE, IERROR 7 INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTENT 9 MPI\_FILE\_GET\_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR) 10INTEGER FH, ETYPE, FILETYPE, IERROR 11 CHARACTER\*(\*) DATAREP 12INTEGER(KIND=MPI\_OFFSET\_KIND) DISP 13 14MPI\_FILE\_IREAD\_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 15<type> BUF(\*) 16INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 17INTEGER(KIND=MPI\_OFFSET\_KIND) OFFSET 18 MPI\_FILE\_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 19 <type> BUF(\*) 20INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 2122 MPI\_FILE\_IREAD\_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 23<type> BUF(\*) 24INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 25MPI\_FILE\_IWRITE\_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 26<type> BUF(\*) 27INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 28 INTEGER(KIND=MPI\_OFFSET\_KIND) OFFSET 2930 MPI\_FILE\_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 31<type> BUF(\*) 32 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 33 MPI\_FILE\_IWRITE\_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 34 <type> BUF(\*) 35 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 36 37 MPI\_FILE\_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) 38 CHARACTER\*(\*) FILENAME 39 INTEGER COMM, AMODE, INFO, FH, IERROR 40 MPI\_FILE\_PREALLOCATE(FH, SIZE, IERROR) 41 INTEGER FH, IERROR 42INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE 43 44MPI\_FILE\_READ\_ALL\_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) 45<type> BUF(\*) 46INTEGER FH, COUNT, DATATYPE, IERROR 47MPI\_FILE\_READ\_ALL\_END(FH, BUF, STATUS, IERROR) 48

1	<type> BUF(*)</type>
3	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
4	MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
5	<type> BUF(*)</type>
6	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
7 8	MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
9	<type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type>
10	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
11 12	MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
12	<type> BUF(*)</type>
14	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
15 16	MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
17	<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>
18 19	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
20	MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
21	<type> BUF(*)</type>
22	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
23	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
24	MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
25	<type> BUF(*)</type>
26 27	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
28	MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
29	<type> BUF(*)</type>
30	INTEGER FH, COUNT, DATATYPE, IERROR
31	MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
32	<type> BUF(*)</type>
33 34	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
35	MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
36	<type> BUF(*)</type>
37	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
38	MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
39	<pre><type> BUF(*)</type></pre>
40	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
41 42	MDT ETLE GEEK(EU NEEGET WUENCE TEDDOD)
42 43	MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR) INTEGER FH, WHENCE, IERROR
43	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45	
46	MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
47	INTEGER FH, WHENCE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
48	INIEGEN(KIND-HEI_OFFGEI_KIND) OFFGEI

MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR	1 $2$
LOGICAL FLAG	3
MPI_FILE_SET_INFO(FH, INFO, IERROR)	4 5
INTEGER FH, INFO, IERROR	6
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)	7
INTEGER FH, IERROR	8
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	9
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)	10
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR	11 12
CHARACTER*(*) DATAREP	13
INTEGER(KIND=MPI_OFFSET_KIND) DISP	14
MPI_FILE_SYNC(FH, IERROR)	15
INTEGER FH, IERROR	16
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	17
<pre><type> BUF(*)</type></pre>	18 19
INTEGER FH, COUNT, DATATYPE, IERROR	20
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)	21
<pre><type> BUF(*)</type></pre>	22
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	23
	24
<pre>MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	25
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	26 27
	28
<pre>MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)</pre>	29
INTEGER FH, COUNT, DATATYPE, IERROR	30
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	31
MOT ETTE UDTTE AT ALL END (EU DILE CTATUS TEDDOD)	32
<pre>MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)</pre>	$33 \\ 34$
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	35
	36
<pre>MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	37
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	38
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	39
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	40 41
<pre><type> BUF(*)</type></pre>	41
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	43
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	44
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	45
<pre><type> BUF(*)</type></pre>	46
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	47
	48

1 2	<pre>MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)</pre>
$\frac{3}{4}$	INTEGER FH, COUNT, DATATYPE, IERROR
5	MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
6	<type> BUF(*)</type>
7	INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
8	MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
9	<type> BUF(*)</type>
10	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
11 12	MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
13	<type> BUF(*)</type>
14	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
15	MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
16	DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
17	CHARACTER*(*) DATAREP
18 19	EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
20	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
21	INTEGER IERROR
22	
23	A.4.12 Language Bindings Fortran Bindings
24	MPI_F_SYNC_REG(buf)
25	<type> buf(*)</type>
26 27	MPI_SIZEOF(X, SIZE, IERROR)
28	<pre><type> X</type></pre>
29	INTEGER SIZE, IERROR
30	MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
31	TYPE(MPI_Status) :: F08_STATUS
32	INTEGER :: F_STATUS(MPI_STATUS_SIZE)
33 34	INTEGER IERROR
35	MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
36	INTEGER :: F_STATUS(MPI_STATUS_SIZE)
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 43	MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR) INTEGER R, NEWTYPE, IERROR
43	
45	MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
46	INTEGER P, R, NEWTYPE, IERROR
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 3
INTEGER LEVEL	4
	5
A.4.14 Deprecated Fortran Bindings	6
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)	7
INTEGER COMM, KEYVAL, IERROR	8 9
	10
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	11
LOGICAL FLAG	12
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)	13
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	14 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)	22
EXTERNAL COPY_FN, DELETE_FN	23
INTEGER KEYVAL, EXTRA_STATE, IERROR	24
MPI_KEYVAL_FREE(KEYVAL, IERROR)	25 26
INTEGER KEYVAL, IERROR	20
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 32
LOGICAL FLAG	32
MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)	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 38
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR	39
LOGICAL FLAG	40
	41
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	42
	43
	44 45
	46
	47

# 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

### B.1.1 Errata to Previous Versions of MPI

1.	Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 25, 176, 538, and	24
	663, and	25
	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	26
	on pages 27, 164, 433, 472 and 513	27
	This is an MPI-2.2 errata: New named predefined datatypes MPI_CXX_BOOL,	28
	MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and	29
	MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding	30
	to the C++ types bool, std::complex <float>, std::complex<double>, and</double></float>	31
	std::complex <long double="">. These datatypes also correspond to the deprecated</long>	32
	C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX,	33
	and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-	34
	standard C++ types Complex<> were substituted by the standard types	35
	<pre>std::complex&lt;&gt;.</pre>	36
		37
2.	Sections 5.9.2 on pages 176 and MPI-2.2 Section 5.9.2, page 165, line 47.	38
	This is an MPI-2.2 errata: MPI_C_COMPLEX was added to the "Complex" reduction	39
	group.	40
3	Section $7.5.5$ on page $302$ , and	41
9.	MPI-2.2, Section 7.5.5 on page 257, $C++$ interface on page 264, line 3.	42
	This is an MPI-2.2 errata: The argument rank was removed and in/outdegree are now	43
	defined as int& indegree and int& outdegree in the C++ interface of	44
	MPI_DIST_GRAPH_NEIGHBORS_COUNT.	45
		46
4.	Section $13.5.2$ , Table $13.2$ on page $538$ , and	47
	MPI-2.2, Section 13.5.3, Table 13.2 on page 433.	48

1 2	This was an MPI-2.2 errata: The MPI_C_BOOL "external 32" representation is corrected to a 1-byte size.
3 4 5 6	<ol> <li>MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 errata: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.</li> </ol>
7 8 9 10 11 12	<ol> <li>Annex A.1.1 on page 663, Table "Optional datatypes (Fortran)", and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37-41. This is an MPI-2.2 errata: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.</li> </ol>
13 14	B.1.2 Changes in MPI-3.0
15 16 17 18 19	<ol> <li>Section 2.6.1 on page 17, Section 16.2 on page 598 and all other chapters. The C++ bindings were removed from the standard. See MPI-2.2 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>
20	This change may anect backward compatibility.
20 21 22 23 24 25 26 27 28 29 30 31	<ol> <li>Section 2.6.1 on page 17, Section 15.1 on page 593 and Section 16.1 on page 597. 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> <li>Section 2.3 on page 10. Clarified parameter usage for IN parameters. C bindings are now const-correct where</li> </ol>
32 33	backward compatibility is preserved.
34 35 36 37	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.
38 39 40 41	<ol> <li>Section 2.5.4 on page 15 and Section 8.1.1 on page 335. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.</li> </ol>
42 43 44 45 46 47 48	<ul> <li>6. Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 16, 25, 27, 176, Sections 4.1, 4.1.7, 4.1.8, 4.1.11, 12.3 on pages 83, 107, 108, 112, 483, and Annex A.1.1 on page 663. New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their results as an MPI_Count value, which is a new type large enough to represent element counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object</li> </ul>

so that a call to MPI\_GET\_ELEMENTS\_X returns the provided MPI\_Count value (in Fortran, INTEGER (KIND=MPI\_COUNT\_KIND). The corresponding predefined datatype is MPI\_COUNT.

- 7. Chapter 3 on page 23 until Chapter 17 on page 599. In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of \*.
- 8. Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 30, 102, 112, 132. The functions MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS were defined to set the count argument to MPI\_UNDEFINED when that argument would overflow. The functions MPI\_PACK\_SIZE and MPI\_TYPE\_SIZE were defined to set the size argument to MPI\_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.
- 9. Section 3.2.6 on page 32, and Section 3.8 on page 64. MPI\_STATUS\_IGNORE can be also used in MPI\_IPROBE, MPI\_PROBE, MPI\_IMPROBE, and MPI\_MPROBE.
- 10. Section 3.8 on page 64 and Section 3.11 on page 81. The use of MPI\_PROC\_NULL in probe operations was clarified. A special predefined message MPI\_MESSAGE\_NO\_PROC was defined for the use of matching probe with MPI\_PROC\_NULL.
- $^{24}$ 11. Sections 3.8.2, 3.8.3, 17.2.4, A.1.1 on pages 67, 69, 648, 663. Like MPI\_PROBE and MPI\_IPROBE, the new MPI\_MPROBE and MPI\_IMPROBE operations allow incoming messages to be queried without actually receiving them, 27except that MPI\_MPROBE and MPI\_IMPROBE provide a mechanism to receive the 28specific message with the new routines MPI\_MRECV and MPI\_IMRECV regardless of 29other intervening probe or receive operations. The opaque object MPI\_Message, the 30 null handle MPI\_MESSAGE\_NULL, and the conversion functions MPI\_Message\_c2f and MPI\_Message\_f2c were defined.
- 12. Section 4.1.2 on page 85 and Section 4.1.13 on page 116. The routine MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK and constant MPI\_COMBINER\_HINDEXED\_BLOCK were added.
- 13. Chapter 5 on page 141 and Section 5.12 on page 196. Added nonblocking interfaces to all collective operations.
- 39 14. Sections 6.4.2, 6.4.4, 11.2.7, on pages 237, 248, 416. 40 The new routines MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO, 41 MPI\_COMM\_GET\_INFO, MPI\_WIN\_SET\_INFO, and MPI\_WIN\_GET\_INFO were added. 42The routine MPI\_COMM\_DUP must also duplicate info hints. 43
- 15. Section 6.4.2 on page 237. Added MPI\_COMM\_IDUP.
- 4616. Section 6.4.2 on page 237. 47Added the new communicator construction routine MPI\_COMM\_CREATE\_GROUP, 48

1  $\mathbf{2}$ 

3

4

5

6

7 8

9

10

11

12

13

14

1516

1718

19

20

21

22

23

25

26

 $^{31}$ 

32 33

34

3536

37

38

44

1 2	which is invoked only by the processes in the group of the new communicator being constructed.
3 4 1 5 6	7. 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.
7 1 8 9 10	<ol> <li>Section 6.6.2 on page 260.</li> <li>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.</li> </ol>
12 <u>1</u> 13 14 15	<ol> <li>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.</li> </ol>
	<ol> <li>Section 7.5.8 on page 312.</li> <li>MPI_CART_MAP can also be used for a zero-dimensional topologies.</li> </ol>
18     2       19     20       21     22       23     24       25     26       27     28       29     29	1. Section 7.6 on page 314 and Section 7.7 on page 324. 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_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.
31 32 33	2. Section 8.7 on page 357 and Section 12.4.3 on page 487. 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 environment by querying the new predefined info object MPI_INFO_ENV.
34 35 2 36	<ol> <li>Section 8.7 on page 357. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.</li> </ol>
37 38 39 40 41	4. Chapter 11 on page 403. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target, new one-sided communication routines, a new memory model, and other changes.
42 2 43 44	5. Section 14.3 on page 563. A new MPI Tool Information Interface was added.
45 46 47 48	The following changes are related to the Fortran language support.

- 26. Section 2.3 on page 10, and Sections 17.1.1, 17.1.2, 17.1.7 on pages 599, 600, and 615. The new mpi\_08 Fortran module was introduced.
- 27. Section 2.5.1 on page 12, Section 17.1.2 on page 600, and Section 17.1.3 on page 603, Section 17.1.7 on page 615.
  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.
- 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 599, 625, 627, 628, 630, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 600, 603, 615.

Within the mpi\_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TR 29113 [42], 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..

- 29. Section 2.6.2 on page 18, Section 17.1.2 on page 600, and Section 17.1.7 on page 615. The ierror dummy arguments are OPTIONAL within the mpi\_08 Fortran module.
- 30. Section 3.2.5 on page 30, Section 17.1.2 on page 600, Section 17.1.3 on page 603, Section 17.1.7 on page 615, and Section 17.2.5 on page 650.
  Within the mpi\_08 Fortran module, the status was defined as TYPE(MPI\_Status). 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.
- 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.
- 32. Section 4.1 on page 83, Section 4.1.6 on page 104, and Section 17.1.15 on page 631. The Fortran alignments of basic datatypes are implementation dependent. It is recommended that they are computed according to BIND(C) derived types. If an array of structures (in C/C++) or derived types (in Fortran) should be communicated, it is recommended that the user creates a portable datatype handle and applies additionally MPI\_TYPE\_CREATE\_RESIZED to this datatype handle.
- 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 111, 183, 189, 276, 281, 343, 345, 347, 593, and 617. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the mpi and mpi\_08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI\_TYPE\_DUP, the Fortran USER\_FUNCTION of MPI\_OP\_CREATE, MPI\_TYPE\_SET\_ATTR, MPI\_TYPE\_SET\_NAME,

 $\mathbf{2}$ 

 $\overline{7}$ 

1 2 3 4 5 6 7 8 9 10 11 12	34.	MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype defini- tion MPI_Type_delete_attr_function, and the predefined callback function MPI_TYPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE, MPI_COMM_CREATE_ERRHANDLER, MPI_WIN_CREATE_ERRHANDLER, MPI_FILE_CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consis- tency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE. Section 6.7.2 on page 267. Section 6.7.2 on page 226. It was clarified that in Fortran, the flag values returned by a comm_copy_attr_fn callback, including MPI_COMM_NULL_COPY_FN and MPI_COMM_DUP_FN, are .FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL.
13 14 15 16 17 18 19 20	35.	Section 8.2 on page 339. With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointer instead of only returning an address-sized integer that may be usable together a with non-standard Cray-pointer. The Fortran interfaces with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR in the mpi module and the mpif.h include file have been deprecated since MPI-3.0.
20 21 22 23	36.	Section 17.1.15 on page 631, and Section 17.1.7 on page 615. Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.
24 25 26 27 28 29 30 31	37.	Section 17.1.16 on page 633 to Section 17.1.19 on page 643, Section 17.1.7 on page 615, and Section 17.1.8 on page 616. The sections about Fortran optimization problems and their solution were partially rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the meaning of the ASYNCHRONOUS attribute were extended to protect nonblocking operations. The Fortran routine MPI_F_SYNC_REG is added. MPI-3.0 compliance for an MPI library together with a Fortran compiler is defined in Section 17.1.7.
32 33 34 35	38.	Section 17.1.2 on page 600. Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.
36 37 38	39.	Section 17.1.3 on page 603, and Section 17.1.7 on page 615. The existing mpi Fortran module must implement compile-time argument checking.
39 40	40.	Section 17.1.4 on page 605. The use of the mpif.h Fortran include file is now strongly discouraged.
41 42 43 44 45	41.	Section A.1.1, Table " <i>Predefined functions</i> " on page 671, Section A.1.3 on page 679, and Section A.3.4 on page 726. Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI) with BIND(C) attribute.
46 47 48	42.	Section A.1.3 on page 679. In some routines, the Fortran callback prototype names were changed fromFN to FUNCTION to be consistent with the other language bindings.

B.2	Changes from Version 2.1 to Version 2.2	1		
1	Section $2.5.4$ on page 15.	2 3		
1.	It is now guaranteed that predefined named constant handles (as other constants)	4		
	can be used in initialization expressions or assignments, i.e., also before the call to	5		
	MPI_INIT.	6		
	-	7		
2.	Section $2.6$ on page $17$ , and Section $16.2$ on page $598$ .	8		
	The C++ language bindings have been deprecated and may be removed in a future	9		
	version of the MPI specification.			
3	Section $3.2.2$ on page $25$ .	11		
5.	MPI_CHAR for printable characters is now defined for C type char (instead of signed	12		
	char). This change should not have any impact on applications nor on MPI libraries	13		
	(except some comment lines), because printable characters could and can be stored in	14		
	any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed	15		
	for predefined reduction operations.	16		
		17		
4.	Section $3.2.2$ on page 25.	18		
	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,	19		
	MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and	20		
	MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.	21		
5	Section 3.4 on page 37, Section 3.7.2 on page 48, Section 3.9 on page 73, and Section 5.1	22		
0.	on page 141.	23		
	The read access restriction on the send buffer for blocking, non blocking and collective	24		
	API has been lifted. It is permitted to access for read the send buffer while the	25		
	operation is in progress.	26		
	operation is in progress.	27		
6.	Section $3.7$ on page $47$ .	28		
	The Advice to users for IBSEND and IRSEND was slightly changed.	29		
7	Section $3.7.3$ on page $52$ .	30		
1.	The advice to free an active request was removed in the Advice to users for	31		
	MPI_REQUEST_FREE.	32		
		33 34		
8.	Section $3.7.6$ on page $63$ .	35		
	MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.	36		
0		37		
9.	Section 5.8 on page 168.	38		
	"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and	39		
	MPI_ALLTOALLW for intracommunicators.	40		
10.	Section $5.9.2$ on page 176.	41		
	Predefined parameterized datatypes (e.g., returned by	42		
	MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g.			
	MPI_REAL8) have been added to the list of valid datatypes in reduction operations.	44		
		45		
11.	Section 5.9.2 on page 176.	46		
	$MPI_(U)INT\{8,16,32,64\}_T$ are all considered C integer types for the purposes of the	47		
	predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran	48		

1 2 3		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.
4 5 6 7	12.	Section 5.9.7 on page 189. The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.
8 9 10 11	13.	Section 5.10.1 on page 190. The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI standard.
11 12 13	14.	Section 5.11.2 on page 194. Added in place argument to MPI_EXSCAN.
14 15 16 17 18 19 20	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.
20 21 22 23 24	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.
25 26 27 28 29 30	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.
31 32 33 34	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.
35 36 37 38	19.	Section 7.5.5 on page 302. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
39 40	20.	Section 8.1.1 on page 335. The subversion number changed from 1 to 2.
41 42 43 44	21.	Section 8.3 on page 342, Section 15.2 on page 596, and Annex A.1.3 on page 679. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.
45 46 47 48	22.	Section 8.7.1 on page 363. 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.

23.	Section 11.3.4 on page 424. The restriction added in MPI 2.1 that the operation MPI_REPLACE in	1 2			
	MPI_ACCUMULATE can be used only with predefined datatypes has been removed.	3			
	MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0.	4			
	Also, a clarification has been made that $MPI\_REPLACE$ can be used only in	5			
	MPI_ACCUMULATE, not in collective operations that do reductions, such as	6			
	MPI_REDUCE and others.	7			
24	Section $12.2$ on page $475$ .	8			
24.	Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for	9 10			
	MPI::Grequest::Start() for consistency with the rest of MPI functions that take function	10			
	pointer arguments.	12			
		13			
25.	Section 13.5.2 on page 536, and Table 13.2 on page 538.	14			
	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX,	15			
	MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX,	16			
	MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes	17			
	in the external 32 representation.	18			
26.	Section 17.2.7 on page 655.	19			
	The description was modified that it only describes how an MPI implementation be-	20			
	haves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example	21			
	16.17 was replaced with three new examples 17.13, 17.14, and 17.15 on pages 656-657	22			
	explicitly detailing cross-language attribute behavior. Implementations that matched	23			
	the behavior of the old example will need to be updated.	24			
27.	Annex A.1.1 on page 663.	25 26			
	Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 678).	20 27			
		28			
28.	Annex A.1.1 on page 663. Table Named Predefined Datatypes.	29			
	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.	32			
_		33			
B.3	Changes from Version 2.0 to Version 2.1	34			
1	Castion 2.2.2 on page 25 and Appen A.1 on page 662	35			
1.	Section 3.2.2 on page 25, and Annex A.1 on page 663. In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn-	36			
	onym for MPI_LONG_LONG_INT.	37			
		38			
2.	Section $3.2.2$ on page $25$ , and Annex A.1 on page $663$ .	39 40			
	MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym),	40			
	MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved	41			
	from optional to official and they are therefore defined for all three language bindings.	43			
3	Section $3.2.5$ on page $30$ .	44			
5.	MPI_GET_COUNT with zero-length datatypes: The value returned as the	45			
	count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes	46			
	have been transferred is zero. If the number of bytes transferred is greater than zero,	47			
	MPI_UNDEFINED is returned.	48			

1 2 3 4 5	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.
6 7 8 9	5.	Section 4.3 on page 138. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
10 11 12 13 14	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).
15 16 17 18	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.
19 20 21	8.	Section 6.7 on page 265. About the attribute caching functions:
22 23 24 25 26 27 28 29		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. ( <i>End of advice to implementors.</i> )
30 31 32 33 34	9.	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.
35 36 37 38	10.	Section 7.4 on page 290. About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must have identical values on all processes of the group of comm_old.
39 40 41 42 43	11.	Section 7.5.1 on page 292. In MPI_CART_CREATE: 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.
44 45 46 47 48	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.

<ul> <li>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.</li> <li>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).</li> <li>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.</li> <li>15. Section 7.5.5 on page 302. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.</li> <li>16. Section 7.5.5 on page 302. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coord sin to significant and 0 is returned in rank.</li> <li>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.</li> <li>18. Section 7.5.7 on page 312. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology.</li> <li>19. Section 8.1.1 on page 335. The subversion number changed from 0 to 1.</li> <li>19. Section 8.1.2 on page 336. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored an name[resulten]. resulten cannot be larger then MPI_MAX_PROCESSOR_NAME.</li> <li>20. Section 8.3 on page 342. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler returned from MPI</li></ul>			
<ul> <li>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>)</li> <li>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. </li> <li>15. Section 7.5.5 on page 302. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank. </li> <li>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. </li> <li>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 Cartesiar communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology. </li> <li>18. Section 7.5.7 on page 312. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology. </li> <li>18.1. Section 8.1.1 on page 335. The subversion number changed from 0 to 1. </li> <li>19. Section 8.1.2 on page 336. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored an amen[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. </li> <li>20. Section 8.3 on page 342. 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_FEE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM_WIN,FILE}_GET_ERRHANDLER to mark </li> </ul>	13.	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	1 2 3 4 5 6
<ul> <li>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.</li> <li>15. Section 7.5.5 on page 302. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.</li> <li>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.</li> <li>17. Section 7.5.6 on page 310. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.</li> <li>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 communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a communicator. This inplies that it is erroneous to call MPI_CART_SHIFT with a communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a communicator. Section 8.1.2 on page 312. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology.</li> <li>18. Section 8.1.1 on page 335. The subversion number changed from 0 to 1.</li> <li>19. Section 8.1.2 on page 336. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored an 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 l</li></ul>		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> )	7 8 9
<ul> <li>In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.</li> <li>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. </li> <li>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. </li> <li>18. Section 7.5.7 on page 312. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology. </li> <li>18. Section 8.1.1 on page 335. The subversion number changed from 0 to 1. </li> <li>19. Section 8.1.2 on page 336. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resulten 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. </li> <li>20. Section 8.3 on page 342. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark </li> </ul>	14.	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	10 11 12 13 14
<ul> <li>In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.</li> <li>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. <li>18. Section 7.5.7 on page 312. 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. </li> <li>18.1. Section 8.1.1 on page 335. The subversion number changed from 0 to 1. </li> <li>19. Section 8.1.2 on page 336. 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. </li> <li>20. Section 8.3 on page 342. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark </li> </li></ul>	15.	In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol-	15 16 17 18
<ul> <li>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.</li> <li>18. Section 7.5.7 on page 312. <ul> <li>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 then newcomm is associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.</li> </ul> </li> <li>18.1. Section 8.1.1 on page 335. <ul> <li>The subversion number changed from 0 to 1.</li> </ul> </li> <li>19. Section 8.1.2 on page 336. <ul> <li>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.</li> </ul> </li> <li>20. Section 8.3 on page 342. <ul> <li>MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to marker.</li> </ul> </li> </ul>	16.	In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian	19 20 21
<ul> <li>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.</li> <li>18.1. Section 8.1.1 on page 335. The subversion number changed from 0 to 1.</li> <li>19. Section 8.1.2 on page 336. 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.</li> <li>20. Section 8.3 on page 342. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark</li> </ul>	17.	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	22 23 24 25 26 27
<ul> <li>The subversion number changed from 0 to 1.</li> <li>19. Section 8.1.2 on page 336. 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.</li> <li>20. Section 8.3 on page 342. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark</li> </ul>	18.	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	28 29 30 31
<ul> <li>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.</li> <li>20. Section 8.3 on page 342.</li> <li>20. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark</li> </ul>	18.1.		32 33 34
<ul> <li>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,</li> <li>MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark</li> </ul>	19.	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	35 36 37 38 39 40
the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.	20.	<ul> <li>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,</li> <li>MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of</li> </ul>	40 41 42 43 44 45 46 47 48

1 2 3 4 5	21.	Section 8.7 on page 357, 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 399.
6 7 8	22.	Section 8.7 on page 357. About MPI_ABORT:
9 10 11 12		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> )
13 14 15 16		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.)
17 18 19 20 21 22 23 23 24	23.	Section 9 on page 367. 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.
25 26 27 28 29	24.	Section 11.3 on page 418. 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.
30 31 32 33 34	25.	Section 11.3 on page 418. 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.
35 36 37	26.	Section 11.3.4 on page 424. MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
38 39 40 41 42 43	27.	Section 13.2.8 on page 500. 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.
44 45 46 47 48	28.	Section 13.2.8 on page 500. 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.

29.	Section 13.3 on page 503. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.	1 2 3
30.	Section 13.5.2 on page 536. The bias of 16 byte doubles was defined with 10383. The correct value is 16383.	4 5 6
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.	7 8 9
32.	Section 17.1.9 on page 617. About MPI_TYPE_CREATE_F90_xxxx:	10 11
	Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_xxxx with the same combination of (xxxx,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same ( REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination ( p,r) in the preceding call to MPI_TYPE_CREATE_F90_xxxx and using a hash- table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (xxxx,p,r). (End of advice to implementors.)	12 13 14 15 16 17 18 19 20 21 22
33.	Section A.1.1 on page 663. MPI_BOTTOM is defined as void * const MPI::BOTTOM.	23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
		47 48

- 1 2

## Bibliography

- V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1
- [4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.
   1.2
- [5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990. 1.2
- [6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. IJHPCN, 1(1/2/3):91–99, 2004. 11.7
- [7] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. 13.1
- [8] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992. 1.2
- [9] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2
- [10] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. 1.2
- [11] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormholerouted networks for interprocessor communication in multicomputers. In *Proceedings* of the 1990 Supercomputing Conference, pages 647–656, 1990. 7.1

9 10

11

12

13 14

15

16

17 18

19

20

21

22

23 24

25

26

27 28

29

30

 $^{31}$ 

32

33 34

35

36 37

38

39

40

41

42

43

 $\frac{44}{45}$ 

46

47

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. CHIMP Concepts, June 1991. 1.2
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

#### Unofficial Draft for Comment Only

- [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, 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. 13.5.2
- [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. 13.5.2

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15 16

17

18

19

20

21

22

23 24

25

26

27 28

29

30

 $^{31}$ 

32

33

34

35

36 37

38

39

40

41

42 43

44

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	[41]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113, Draft N1869. TR on further interoperability with C, July, 18 2011. http://www.nag.co.uk/sc22wg5/ and ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1869.pdf. 17.1.2
13 14 15	[42]	International Organization for Standardization, ISO/IEC/SC22/WG5 (For- tran), Geneva, TR 29113. TR on further interoperability with C, 2012. http://www.nag.co.uk/sc22wg5/. 17.1.1, 17.1.2, 17.1.7, 28
16 17 18	[43]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 4.1.4
19 20 21 22	[44]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994</i> Symposium on Operating Systems Design and Implementation, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1
23 24	[45]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 7.1
25 26 27 28 29 30	[46]	S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at http://www.netbsd.org/Documentation/lite2/psd/. 10.5.5
31 32	[47]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2
33 34	[48]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1
35 36 37	[49]	William J. Nitzberg. <i>Collective Parallel I/O</i> . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1
38 39 40	[50]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 10.5.5
41 42 43	[51]	Paul Pierce. The NX/2 operating system. In <i>Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications</i> , pages 384–390. ACM Press, 1988. 1.2
44 45 46 47 48	[52]	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

 $\mathbf{2}$ 

 $^{31}$ 

- [53] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. 13.1
- [54] 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
- [55] 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
- [56] 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
- [57] 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
- [58] 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
- [59] 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
- [60] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2
- [61] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

### **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 examples.

14ASYNCHRONOUS, 637, 645 15Attributes between languages, 656 1617Basic tool using performance variables in the MPI tool information interface, 18 58419 20C/Fortran handle conversion, 649 21Cartesian virtual topologies, 330 22 Client-server code, 62, 63 23with blocking probe, 6624with blocking probe, wrong, 662526Datatype 273D array, 123 28absolute addresses, 12829 array of structures, 125 30 elaborate example, 136, 137  $^{31}$ matching type, 11232 matrix transpose, 124, 125 33 union, 13034Datatypes 35 matching, 33 36 not matching, 3437 untyped, 3438 Deadlock 39 if not buffered, 4340 with MPI\_Bcast, 214, 215 41 wrong message exchange, 43 4243 False matching of collective operations, 218 44Fortran 90 copying and sequence problem, 45627, 629, 630 46Fortran 90 derived types, 631 47Fortran 90 heterogeneous communication, 48 624.625

Fortran 90 invalid KIND, 620 Fortran 90 MPI\_TYPE\_MATCH\_SIZE implementation, 624 Fortran 90 overlapping communication and computation, 641, 642, 646 Fortran 90 register optimization, 634–636 Independence of nonblocking operations, 221Intercommunicator, 241, 245 Interlanguage communication, 660 Intertwined matching pairs, 41 Message exchange, 42Mixing blocking and nonblocking collective operations, 217 Mixing collective and point-to-point requests, 220 MPI\_ACCUMULATE, 426 MPI\_Accumulate, 468, 470 MPI\_Aint, 125 MPI\_Allgather, 167 MPI\_ALLOC\_MEM, 341 MPI\_Alloc\_mem, 341, 470 MPI\_ALLREDUCE, 188 MPI\_Alltoall, 219 MPI\_Barrier, 361, 458–461, 467–469 MPI\_Bcast, 149, 214-218 MPI\_BSEND, 41 MPI\_Buffer\_attach, 45, 360 MPI\_Buffer\_detach, 45 MPI\_BYTE, 34 MPI\_Cancel, 361 MPI\_CART\_COORDS, 311 MPI\_CART\_GET, 330

MPI_CART_RANK, 311	MPI_Group_excl, 251	1
MPI_CART_SHIFT, 311, 330	MPI_Group_free, 241, 251, 252	2
MPI_CART_SUB, 312	MPI_Group_incl, 241, 252, 255	3
MPI_CHARACTER, 35	MPI_Iallreduce, 220	4
MPI_Comm_create, 241, 251, 252, 255	MPI_Ialltoall, 219	5
MPI_Comm_create_keyval, 279	MPI_Ibarrier, 217–220	6
MPI_Comm_dup, 254	MPI_Ibcast, 200, 220, 221	7
MPI_Comm_get_attr, 279	MPI_INFO_ENV, 359	8
MPI_Comm_group, 241, 255, 279	MPI_Intercomm_create, 263, 264	9
MPI_Comm_remote_size, 245	MPI_Iprobe, 361	10
MPI_Comm_set_attr, 279	MPI_IRECV, 54–56, 62, 63	11
MPI_COMM_SPAWN, 378	MPI_Irecv, 220	12
MPI_Comm_spawn, 378	MPI_ISEND, 54–56, 62, 63	13
MPI_COMM_SPAWN_MULTIPLE, 384	MPI_Op_create, 186, 187, 195	14
MPI_Comm_spawn_multiple, 384	MPI_Pack, 135–137	15
MPI_Comm_split, 245, 263, 264	MPI_Pack_size, 137	16
MPI_Compare_and_swap, 469, 470	MPI_PROBE, 66	17
MPI_DIMS_CREATE, 293, 330	MPI_Put, 443, 448, 458, 459, 461, 465, 466	18
MPI_DIST_GRAPH_CREATE, 300	MPI_RECV, 33–35, 41–43, 56, 66, 112	19
MPI_Dist_graph_create, 301	MPI_Recv, 219	20
MPI_DIST_GRAPH_CREATE_ADJACENT,	·	21
300	MPI_Reduce, 180, 182, 186, 187	22
MPI_FILE_CLOSE, 514, 517	MPI_REQUEST_FREE, 55	23
MPI_FILE_GET_AMODE, 499	MPI_Request_free, 360	24
MPI_FILE_IREAD, 517	MPI_Rget, 469	25
MPI_FILE_OPEN, 514, 517	MPI_Rput, 469	26
MPI_FILE_READ, 514	MPI_Scan, 195	27
MPI_FILE_SET_ATOMICITY, 548	MPI_Scatter, 162	28
MPI_FILE_SET_VIEW, 514, 517	MPI_Scatterv, 162, 163	29
MPI_FILE_SYNC, 549	MPI_SEND, 33–35, 42, 43, 56, 66, 112	30
MPI_Finalize, 360, 361	MPI_Send, 125, 128, 130, 136, 219, 220	31
MPI_FREE_MEM, 341	MPI_SENDRECV, 123–125	32
MPI_Free_mem, 470	MPI_SENDRECV_REPLACE, 311	33
MPI_Gather, 137, 152, 153, 157	MPI_SSEND, 41, 56	34
MPI_Gatherv, 137, 154–157	MPI_Test_cancelled, 361	35
MPI_GET, 422, 423	MPI_TYPE_COMMIT, 110, 123–125, 422,	36
MPI_Get, 458–460, 465–467	631, 632	37
MPI_Get_accumulate, 468, 470	MPI_Type_commit, 125, 128, 130, 136,	38
MPI_GET_ADDRESS, 103, 631, 632, 653	153-157, 163, 195	39
MPI_Get_address, 125, 128, 130, 136	MPI_TYPE_CONTIGUOUS, 86, 105, 112,	40
MPI_GET_COUNT, 114	114	41
MPI_GET_ELEMENTS, 114	MPI_Type_contiguous, 153	42
MPI_GRAPH_CREATE, 294, 307	MPI_TYPE_CREATE_DARRAY, 101	43
MPI_GRAPH_NEIGHBORS, 307	MPI_TYPE_CREATE_HVECTOR, 123,	44
MPI_GRAPH_NEIGHBORS_COUNT,	124	45
307	MPI_Type_create_hvector, 125, 128	46
MPI_Grequest_complete, 480	MPI_TYPE_CREATE_INDEXED_BLOCK,	47
MPI_Grequest_start, 480	422	48

1 MPI\_TYPE\_CREATE\_RESIZED, 631,  $\mathbf{2}$ 3 MPI\_TYPE\_CREATE\_STRUCT, 94, 105, 4 125, 631, 632  $\mathbf{5}$ MPI\_Type\_create\_struct, 125, 128, 130, 6 136, 156, 157, 195  $\overline{7}$ MPI\_TYPE\_CREATE\_SUBARRAY, 555 8 MPI\_TYPE\_EXTENT, 422 9 MPI\_TYPE\_FREE, 422 10 MPI\_Type\_get\_contents, 130 11 MPI\_Type\_get\_envelope, 130 12MPI\_TYPE\_GET\_EXTENT, 123-125, 13423.426 14MPI\_Type\_get\_extent, 125 15MPI\_TYPE\_INDEXED, 89, 124 16MPI\_Type\_indexed, 125, 128 17MPI\_TYPE\_VECTOR, 87, 123-125 18MPI\_Type\_vector, 154, 155, 157, 163 19MPI\_Unpack, 136, 137 20MPI\_User\_function, 187 21MPI\_WAIT, 54-56, 62, 63, 517 22MPI\_Wait, 217-220 23MPI\_Waitall, 220, 469  $^{24}$ MPI\_WAITANY, 62 25MPI\_Waitany, 469 26MPI\_WAITSOME, 63 27MPI\_Win\_attach, 470 28MPI\_Win\_complete, 443, 460, 461, 466, 29467 30 MPI\_WIN\_CREATE, 422, 423, 426  $^{31}$ MPI\_Win\_create\_dynamic, 470 32MPI\_Win\_detach, 470 33 MPI\_WIN\_FENCE, 422, 423, 426 34MPI\_Win\_fence, 465 35MPI\_Win\_flush, 459, 468-470 36 MPI\_Win\_flush\_all, 468 37MPI\_Win\_flush\_local, 458 38MPI\_WIN\_FREE, 423, 426 39MPI\_Win\_lock, 448, 458-461 40MPI\_Win\_lock\_all, 469, 470  $^{41}$ MPI\_Win\_post, 460, 461, 466, 467 42MPI\_Win\_start, 443, 460, 461, 466, 467 43MPI\_Win\_sync, 458, 459, 468, 469 44MPI\_Win\_unlock, 448, 458-461 45MPI\_Win\_unlock\_all, 469, 470 46MPI\_Win\_wait, 460, 461, 466, 467 47mpiexec, 359, 365, 366 48

Neighborhood collective communication, 330 No Matching of Blocking and Nonblocking collective operations, 219 Non-deterministic program with MPI\_Bcast, 216 Non-overtaking messages, 41 Nonblocking operations, 54, 55 message ordering, 56 progress, 56 Overlapping Communicators, 220 Pipelining nonblocking collective operations, 220 Profiling interface, 560

Progression of nonblocking collective operations, 219

Reading the value of a control variable in the MPI tool information interface, 574

Threads and MPI, 485Topologies, 330 Typemap, 85–87, 89, 94, 101

Using MPI\_T\_CVAR\_GET\_INFO to list all names of control variables., 571

Virtual topologies, 330

## MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI::\_LONG\_LONG, 790 MPI::BOOL, 789 MPI::COMBINER\_HINDEXED\_INTEGER, 598 MPI::COMBINER\_HVECTOR\_INTEGER, 598 MPI::COMBINER\_STRUCT\_INTEGER, 598 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::LB, 598 MPI::LONG\_DOUBLE\_COMPLEX, 789 MPI::LONG\_LONG, 790 MPI::REAL16, 790 MPI::UB, 598 MPI\_2DOUBLE\_PRECISION, 180, 669 MPI\_2INT, 180, 669 MPI\_2INTEGER, 180, 669 MPI\_2REAL, 180, 669 MPI\_ADDRESS\_KIND, 15, 16, 16, 26, 266, 627, 656, 666 MPI\_AINT, 25, 27, 177, 667, 668, 795, 797 MPI\_ANY\_SOURCE, 28, 29, 41, 51, 52, 64, 65, 67-69, 76, 79, 80, 287, 337, 665 MPI\_ANY\_TAG, 15, 28, 29, 31, 51, 52, 64, 65, 67–71, 76, 79–81, 665, 792 MPI\_APPNUM, 398, 399, 673 MPI\_ARGV\_NULL, 16, 378, 379, 626, 675 MPI\_ARGVS\_NULL, 16, 383, 626, 675

MPI_ASYNC_PROTECTS_NONBLOCKING	13
15, 600, 601, 604, 606, 607, 616,	14
617, 638, 666, 794	15
MPI_BAND, 176, 177, 670	16
MPI_BOR, 176, 177, 670	17
MPI_BOTTOM, 10, 16, 32, 102, 115, 116,	18
145, 298, 300, 380, 412, 416, 602,	19
605, 611, 626, 631, 633, 635, 636,	20
638-640, 643, 653, 654, 660, 665,	21
801	22
MPI_BSEND_OVERHEAD, 46, 665	23
MPI_BXOR, 176, 177, 670	24
MPI_BYTE, 25, 26, 33, 34, 36, 138, 177,	25
492, 534, 546, 660, 667, 668, 798	26
MPI_C_BOOL, 26, 177, 667, 790, 795–797	27
MPI_C_COMPLEX, 26, 177, 667, 789,	28
795-797	29
MPI_C_DOUBLE_COMPLEX, 26, 177,	30
667, 795–797	31
MPI_C_FLOAT_COMPLEX, 177, 667,	32
795–797	33
MPI_C_LONG_DOUBLE_COMPLEX,	34
26, 177, 667, 795-797	35
MPI_CART, 303, 671	36
MPI_CHAR, 26, 36, 94, 178, 179, 567, 667,	37
795	38
MPI_CHARACTER, 25, 34–36, 178, 179,	39
668	40
MPI_COMBINER_CONTIGUOUS, 118,	41
121, 674	42
MPI_COMBINER_DARRAY, 118, 122,	43
674	44
MPI_COMBINER_DUP, 118, 121, 674	45
MPI_COMBINER_F90_COMPLEX, 118,	46
123, 674	47
,	48

> 5 6

7

9 10 11

1	MPI_COMBINER_F90_INTEGER, 118,
2	123,674
3	MPI_COMBINER_F90_REAL, 118, 122,
4 5	674 MDI COMDINED HINDEVED 10, 110
6	MPI_COMBINER_HINDEXED, 18, 118, 121, 674
7	MPI_COMBINER_HINDEXED_BLOCK,
8	118, 122, 674, 791
9	MPI_COMBINER_HINDEXED_INTEGER,
10	18, 598, 790
11	MPI_COMBINER_HVECTOR, 18, 118,
12	121, 674
13 14	MPI_COMBINER_HVECTOR_INTEGER,
14	18, 598, 790 MPI_COMBINER_INDEXED, 118, 121,
16	674
17	MPI_COMBINER_INDEXED_BLOCK,
18	118, 121, 674
19	MPI_COMBINER_NAMED, 117, 118,
20	121,674
21	MPI_COMBINER_RESIZED, 118, 123,
22	674
23 24	MPI_COMBINER_STRUCT, 18, 118,
24 25	122, 674 MPI_COMBINER_STRUCT_INTEGER,
26	18, 598, 790
27	MPI_COMBINER_SUBARRAY, 118, 122,
28	674
29	MPI_COMBINER_VECTOR, 118, 121,
30	674
31	MPI_COMM_DUP_FN, 18, <u>269</u> , 609, 672,
32	794
33 34	MPI_COMM_NULL, 227, 240, 241,
35	243–245, 247, 248, 283, 292, 294, 381, 400–402, 671, 798
36	MPI_COMM_NULL_COPY_FN, 18, <u>269</u> ,
37	602, 654, 672, 794
38	MPI_COMM_NULL_DELETE_FN, 18,
39	269, 672
40	MPI_COMM_PARENT, 283
41	MPI_COMM_SELF, 227, 243, 266, 283,
42	363, 400, 494, 669, 796
43	MPI_COMM_TYPE_SHARED, 248, 669,
44 45	792 MDL COMM WORLD 15 22 27 28
45	MPI_COMM_WORLD, 15, 22, 27, 28, 227–229, 236, 237, 252, 261, 283,
47	237-229, 230, 237, 232, 201, 203, 293, 336-338, 342, 344, 353,
48	360-362, 364, 373, 374, 376, 377,

381, 383, 397-400, 488, 533, 552, 573, 580, 648, 659, 669, 800 MPI\_COMPLEX, 25, 177, 536, 618, 668 MPI\_COMPLEX16, 177, 668 MPI\_COMPLEX32, 177, 668 MPI\_COMPLEX4, 177, 668 MPI\_COMPLEX8, 177, 668 MPI\_CONGRUENT, 237, 260, 669 MPI\_CONVERSION\_FN\_NULL, 541, 672 MPI\_COUNT, 25, 27, 177, 567, 667, 668, 791 MPI\_COUNT\_KIND, 15, 26, 666 MPI\_CXX\_BOOL, 27, 177, 668, 789 MPI\_CXX\_DOUBLE\_COMPLEX, 27, 177, 668, 789 MPI\_CXX\_FLOAT\_COMPLEX, 27, 177, 668, 789 MPI\_CXX\_LONG\_DOUBLE\_COMPLEX, 27, 177, 668, 789 MPI\_DATATYPE\_NULL, 111, 671 MPI\_DISPLACEMENT\_CURRENT, 504, 675, 801 MPI\_DIST\_GRAPH, 303, 671, 796 MPI\_DISTRIBUTE\_BLOCK, 99, 675 MPI\_DISTRIBUTE\_CYCLIC, 99, 675 MPI\_DISTRIBUTE\_DFLT\_DARG, 99, 675 MPI\_DISTRIBUTE\_NONE, 99, 675 MPI\_DOUBLE, 26, 176, 567, 577, 617, 667 MPI\_DOUBLE\_COMPLEX, 25, 177, 536, 618, 668 MPI\_DOUBLE\_INT, 180, 669 MPI\_DOUBLE\_PRECISION, 25, 176, 618,668 MPI\_DUP\_FN, 18, 269, <u>594</u>, 673 MPI\_ERR\_ACCESS, 351, 497, 553, 664 MPI\_ERR\_AMODE, 351, 495, 553, 664 MPI\_ERR\_ARG, 350, 663 MPI\_ERR\_ASSERT, 350, 453, 664 MPI\_ERR\_BAD\_FILE, 351, 553, 664 MPI\_ERR\_BASE, 340, 350, 453, 664 MPI\_ERR\_BUFFER, 350, 663 MPI\_ERR\_COMM, 350, 663 MPI\_ERR\_CONVERSION, 351, 542, 553, 664 MPI\_ERR\_COUNT, 350, 663 MPI\_ERR\_DIMS, 350, 663 MPI\_ERR\_DISP, 350, 453, 664

Unofficial Draft for Comment Only

MPI\_ERR\_DUP\_DATAREP, 351, 539, 553, 664 MPI\_ERR\_FILE, 351, 553, 664 MPI\_ERR\_FILE\_EXISTS, 351, 553, 664 MPI\_ERR\_FILE\_IN\_USE, 351, 497, 553, 664 MPI\_ERR\_GROUP, 350, 663 MPI\_ERR\_IN\_STATUS, 30, 32, 53, 59, 61, 344, 350, 479, 509, 664 MPI\_ERR\_INFO, 350, 664 MPI\_ERR\_INFO\_KEY, 350, 368, 664 MPI\_ERR\_INFO\_NOKEY, 350, 369, 664 MPI\_ERR\_INFO\_VALUE, 350, 368, 664 MPI\_ERR\_INTERN, 350, 663 MPI\_ERR\_IO, 351, 553, 664 MPI\_ERR\_KEYVAL, 279, 350, 664 MPI\_ERR\_LASTCODE, 349, 351, 353, 354, 590, 665 MPI\_ERR\_LOCKTYPE, 350, 453, 664 MPI\_ERR\_NAME, 350, 394, 664 MPI\_ERR\_NO\_MEM, 340, 350, 664 MPI\_ERR\_NO\_SPACE, 351, 553, 664 MPI\_ERR\_NO\_SUCH\_FILE, 351, 496, 553.664 MPI\_ERR\_NOT\_SAME, 351, 553, 664 MPI\_ERR\_OP, 350, 453, 663 MPI\_ERR\_OTHER, 350, 352, 663 MPI\_ERR\_PENDING, 59, 350, 663 MPI\_ERR\_PORT, 350, 391, 664 MPI\_ERR\_QUOTA, 351, 553, 664 MPI\_ERR\_RANK, 350, 453, 663 MPI\_ERR\_READ\_ONLY, 351, 553, 664 MPI\_ERR\_REQUEST, 350, 663 MPI\_ERR\_RMA\_ATTACH, 351, 453, 664 MPI\_ERR\_RMA\_CONFLICT, 350, 453, 664 MPI\_ERR\_RMA\_RANGE, 351, 453, 664 MPI\_ERR\_RMA\_SHARED, 351, 453, 664 MPI\_ERR\_RMA\_SYNC, 350, 453, 664 MPI\_ERR\_RMA\_WRONG\_FLAVOR, 351, 411, 453, 664 MPI\_ERR\_ROOT, 350, 663 MPI\_ERR\_SERVICE, 350, 393, 664 MPI\_ERR\_SIZE, 350, 453, 664 MPI\_ERR\_SPAWN, 350, 379, 380, 664 MPI\_ERR\_TAG, 350, 663 MPI\_ERR\_TOPOLOGY, 350, 663 MPI\_ERR\_TRUNCATE, 350, 663

MPI\_ERR\_TYPE, 350, 663  $\mathbf{2}$ MPI\_ERR\_UNKNOWN, 350, 352, 663 MPI\_ERR\_UNSUPPORTED\_DATAREP, 351, 553, 664 MPI\_ERR\_UNSUPPORTED\_OPERATION, 5351, 553, 664 MPI\_ERR\_WIN, 350, 453, 664 7 MPI\_ERRCODES\_IGNORE, 16, 380, 626, 9 675 MPI\_ERRHANDLER\_NULL, 349, 671 10 MPI\_ERROR, 30, 53, 197, 431, 666 11 12MPI\_ERRORS\_ARE\_FATAL, 342, 343, 13 355, 452, 552, 666 MPI\_ERRORS\_RETURN, 342, 343, 356, 1415552, 659, 666 16MPI\_F08\_STATUS\_IGNORE, 651, 676, 17793 18 MPI\_F08\_STATUSES\_IGNORE, 651, 676, 19793 MPI\_F\_STATUS\_IGNORE, 650, 676 20MPI\_F\_STATUSES\_IGNORE, 650, 676 21MPI\_FILE\_NULL, 496, 552, 671 22MPI\_FLOAT, 26, 94, 174, 176, 535, 667 23MPI\_FLOAT\_INT, 12, 180, 669  $^{24}$ MPI\_GRAPH, 303, 671 2526MPI\_GROUP\_EMPTY, 226, 232, 240, 241, 243, 671 27MPI\_GROUP\_NULL, 226, 235, 671 28MPI\_HOST, 337, 670 29MPI\_IDENT, 229, 237, 669 30 MPI\_IN\_PLACE, 16, 144, 171, 605, 626, 3132 665 33 MPI\_INFO\_ENV, 358, 359, 669, 792 MPI\_INFO\_NULL, 300, 372, 380, 389, 34 495, 496, 505, 671 35 MPI\_INT, 12, 26, 84, 176, 535, 536, 567, 36 570, 575, 579, 617, 659, 661, 667 37 MPI\_INT16\_T, 26, 176, 667, 795, 797 38 MPI\_INT32\_T, 26, 176, 667, 795, 797 39 MPI\_INT64\_T, 26, 176, 667, 795, 797 40 MPI\_INT8\_T, 26, 176, 667, 795, 797 41 42MPI\_INTEGER, 25, 33, 176, 617, 618, 661,668 43 MPI\_INTEGER1, 25, 176, 668 44MPI\_INTEGER16, 176, 668 4546MPI\_INTEGER2, 25, 176, 536, 668 MPI\_INTEGER4, 25, 176, 668 4748 MPI\_INTEGER8, 176, 622, 668

1	MPI_INTEGER_KIND, 15, 666
2	MPI_IO, 337, 670
3	MPI_KEYVAL_INVALID, 270–272, 665
4	MPI_LAND, 176, 177, 670
5	MPI_LASTUSEDCODE, 353, 673
6	MPI_LB, 18, 598, 790
7	MPI_LOCK_EXCLUSIVE, 446, 665
8	MPI_LOCK_SHARED, 446, 447, 665
9	MPI_LOGICAL, 25, 177, 668
10	MPI_LONG, 26, 176, 667
11	MPI_LONG_DOUBLE, 26, 176, 667
12	MPI_LONG_DOUBLE_INT, 180, 669
13	MPI_LONG_INT, 180, 669
14	MPI_LONG_LONG, 26, 176, 667, 797
15	MPI_LONG_LONG_INT, 26, 176, 667,
16	797
17	MPI_LOR, 176, 177, 670
18	MPI_LOR, 176, 177, 670
19	
20	MPI_MAX, 174, 176, 177, 195, 670
20	MPI_MAX_DATAREP_STRING, 15, 506,
21	539, 666
22	MPI_MAX_ERROR_STRING, 15, 349,
23 24	354, 666
	MPI_MAX_INFO_KEY, 15, 350, 367, 370,
25 26	666 NDL MAX INFO MAL 15 250 205 600
26	MPI_MAX_INFO_VAL, 15, 350, 367, 666
27	MPI_MAX_LIBRARY_VERSION_STRING,
28	15, 336, 666, 790
29	MPI_MAX_OBJECT_NAME, 15,
30	282–284, 666, 792, 798
31	MPI_MAX_PORT_NAME, 15, 389, 666
32	MPI_MAX_PROCESSOR_NAME, 15,
33	338, 339, 666, 799
34	MPI_MAXLOC, 176, 179, 180, 183, 670
35	MPI_MESSAGE_NO_PROC, 69–71, 665,
36	791
37	MPI_MESSAGE_NULL, 69–71, 671, 791
38	MPI_MIN, 176, 177, 670
39	MPI_MINLOC, 176, 179, 180, 183, 670
40	MPI_MODE_APPEND, 494, 495, 674
41	MPI_MODE_CREATE, 494, 495, 503, 674
42	MPI_MODE_DELETE_ON_CLOSE,
43	494 - 496, 674
44	MPI_MODE_EXCL, 494, 495, 674
45	MPI_MODE_NOCHECK, 447, 451, 452,
46	674
47	MPI_MODE_NOPRECEDE, 451, 452, 674
48	MPI_MODE_NOPUT, 451, 452, 674

MPI\_MODE\_NOSTORE, 451, 452, 674 MPI\_MODE\_NOSUCCEED, 451, 452, 674 MPI\_MODE\_RDONLY, 494, 495, 500, 674 MPI\_MODE\_RDWR, 494, 495, 674 MPI\_MODE\_SEQUENTIAL, 494, 495, 497, 498, 504, 509, 514, 524, 545, 674, 801 MPI\_MODE\_UNIQUE\_OPEN, 494, 495, 674 MPI\_MODE\_WRONLY, 494, 495, 674 MPI\_NO\_OP, 428, 429, 670 MPI\_NULL\_COPY\_FN, 18, 269, 594, 673 MPI\_NULL\_DELETE\_FN, 18, 269, 594, 673 MPI\_OFFSET, 25, 177, 667, 668, 795, 797 MPI\_OFFSET\_KIND, 15, 16, 26, 547, 627,666 MPI\_OP\_NULL, 186, 671 MPI\_ORDER\_C, 14, 96, 99, 100, 675 MPI\_ORDER\_FORTRAN, 14, 96, 99, 675 MPI\_PACKED, 12, 25, 26, 33, 133, 134, 138, 537, 660, 667, 668 MPI\_PROC\_NULL, 24, 30, 65, 69-71, 81, 146, 148, 150, 152, 160, 162, 175, 229, 311, 315, 337, 411, 419, 665, 791, 798, 800 MPI\_PROD, 176, 177, 670 MPI\_REAL, 25, 33, 176, 536, 617, 618, 625,668 MPI\_REAL16, 177, 668 MPI\_REAL2, 25, 177, 668 MPI\_REAL4, 25, 177, 617, 622, 668 MPI\_REAL8, 25, 177, 617, 668, 795 MPI\_REPLACE, 426-429, 469, 670, 797, 800 MPI\_REQUEST\_NULL, 52–55, 57–61, 478, 671 MPI\_ROOT, 146, 665 MPI\_SEEK\_CUR, 519, 525, 675 MPI\_SEEK\_END, 519, 525, 675 MPI\_SEEK\_SET, 519, 525, 675 MPI\_SHORT, 26, 176, 667 MPI\_SHORT\_INT, 180, 669 MPI\_SIGNED\_CHAR, 26, 176, 178, 179, 667, 797 MPI\_SIMILAR, 229, 237, 260, 669 MPI\_SOURCE, 30, 197, 666

MPI_STATUS_IGNORE, 10, 16, 32, 477,	
509,605,626,650,651,660,675,	
676, 791	
MPI_STATUS_SIZE, 15, 30, 606, 666	
MPI_STATUSES_IGNORE, 14, 16, 32,	
477, 479, 626, 650, 651, 675, 676	
MPI_SUBARRAYS_SUPPORTED, 15,	
600,  601,  604,  606-609,  612-615,	
628,629,666,793	
MPI_SUBVERSION, 336, 676	
MPI_SUCCESS, 19, 52, 59, 61, 269–274,	
276, 277, 349, 350, 352, 355, 356,	
380, 542, 563, 581 - 583, 590, 591,	
594,663	
MPI_SUM, 176, 177, 659, 670	
MPI_T_BIND_MPI_COMM, 565, 677	
MPI_T_BIND_MPI_DATATYPE, 565,	
677	
MPI_T_BIND_MPI_ERRHANDLER,	
565,677	
MPI_T_BIND_MPI_FILE, 565, 677	
MPI_T_BIND_MPI_GROUP, 565, 677	
MPI_T_BIND_MPI_INFO, 565, 677	
MPI_T_BIND_MPI_MESSAGE, 565, 677	
MPI_T_BIND_MPI_OP, 565, 677	
MPI_T_BIND_MPI_REQUEST, 565, 677	
MPI_T_BIND_MPI_WIN, 565, 677	
MPI_T_BIND_NO_OBJECT, 565, 571,	
573, 579–581, 677	
MPI_T_CVAR_HANDLE_NULL, 573, 676	
MPI_T_ENUM_NULL, 570, 579, 676	
MPI_T_ERR_CANNOT_INIT, 591, 665	
MPI_T_ERR_CVAR_SET_NEVER, 574,	
591, 665	
MPI_T_ERR_CVAR_SET_NOT_NOW,	
574, 591, 665	
MPI_T_ERR_INVALID_HANDLE, 591,	
665 MDI T EDD INVALID INDEX 501 665	
MPI_T_ERR_INVALID_INDEX, 591, 665	
MPI_T_ERR_INVALID_ITEM, 591, 665	
MPI_T_ERR_INVALID_SESSION, 591, 665	
MPI_T_ERR_MEMORY, 591, 665 MPI_T_ERR_NOT_INITIALIZED, 591,	
665 MP1_1_ERR_NO1_INTIALIZED, 591,	
MPI_T_ERR_OUT_OF_HANDLES, 591,	
$\begin{array}{c} \text{MF1_1}_\text{ERR_OU1_OF_MANDLES, 591,} \\ 665 \end{array}$	
000	

MPI_T_ERR_OUT_OF_SESSIONS, 591,	1
665	2
MPI_T_ERR_PVAR_NO_ATOMIC, 583,	3
591,665	4
MPI_T_ERR_PVAR_NO_STARTSTOP,	5
581, 582, 591, 665	6
MPI_T_ERR_PVAR_NO_WRITE, 583,	7
591, 665	8
,	9
MPI_T_PVAR_ALL_HANDLES, 581–584,	10
677	
MPI_T_PVAR_CLASS_AGGREGATE,	11
577,678	12
MPI_T_PVAR_CLASS_COUNTER, 576,	13
678	14
MPI_T_PVAR_CLASS_GENERIC, 577,	15
678	16
MPI_T_PVAR_CLASS_HIGHWATERMARK.	,17
576, 678	18
MPI_T_PVAR_CLASS_LEVEL, 576, 678	19
MPI_T_PVAR_CLASS_LOWWATERMARK,	20
576, 678	21
MPI_T_PVAR_CLASS_PERCENTAGE,	22
576, 678	23
MPI_T_PVAR_CLASS_SIZE, 576, 678	24
MPI_T_PVAR_CLASS_SIZE, 576, 678 MPI_T_PVAR_CLASS_STATE, 575, 678	25
, , ,	26
MPI_T_PVAR_CLASS_TIMER, 577, 678	27
MPI_T_PVAR_HANDLE_NULL, 581, 676	
MPI_T_PVAR_SESSION_NULL, 580, 676	28
MPI_T_SCOPE_ALL, 571, 677	29
MPI_T_SCOPE_ALL_EQ, 571, 574, 677	30
MPI_T_SCOPE_CONSTANT, 571, 677	31
MPI_T_SCOPE_GROUP, 571, 677	32
MPI_T_SCOPE_GROUP_EQ, 571, 574,	33
677	34
MPI_T_SCOPE_LOCAL, 571, 677	35
MPI_T_SCOPE_READONLY, 571, 677	36
MPI_T_VERBOSITY_MPIDEV_ALL,	37
564, 676	38
MPI_T_VERBOSITY_MPIDEV_BASIC,	39
564, 676	40
MPI_T_VERBOSITY_MPIDEV_DETAIL,	41
564, 676	42
MPI_T_VERBOSITY_TUNER_ALL, 564,	43
676	44
	45
MPI_T_VERBOSITY_TUNER_BASIC,	46
564, 676	40 47
MPI_T_VERBOSITY_TUNER_DETAIL,	
564,676	48

1	
2	MPI_T_VERBOSITY_USER_ALL, 564,
3 4	MPI_T_VERBOSITY_USER_BASIC,
4 5	564, 676
6	MPI_T_VERBOSITY_USER_DETAIL,
	564, 676
7	MPI_TAG, 30, 197, 666
8	MPI_TAG_UB, 27, 337, 655, 658, 670
9	MPI_THREAD_FUNNELED, 488, 674
10	MPI_THREAD_MULTIPLE, 488, 490,
11	674
12	MPI_THREAD_SERIALIZED, 488, 674
13	MPI_THREAD_SINGLE, 488, 489, 674
14	MPI_TYPE_DUP_FN, <u>276</u> , 672
15	MPI_TYPE_NULL_COPY_FN, <u>276</u> , 672
16	MPI_TYPE_NULL_DELETE_FN, <u>276</u> ,
17	672, 794
18	MPI_TYPECLASS_COMPLEX, 623, 675
19	MPI_TYPECLASS_INTEGER, 623, 675
20	MPI_TYPECLASS_REAL, 623, 675
21	MPI_UB, 4, 18, 598, 790
22	MPI_UINT16_T, 26, 176, 667, 795, 797
23	MPI_UINT32_T, 26, 176, 667, 795, 797
24	MPI_UINT64_T, 26, 176, 667, 795, 797
25	MPI_UINT8_T, 26, 176, 667, 795, 797
26	MPI_UNDEFINED, 31, 58, 61, 62, 104,
27 28	107, 109, 114, 135, 228, 229, 245,
28 29	303, 313, 314, 619, 620, 665, 791,
30	797
31	MPI_UNEQUAL, 229, 237, 259, 669
32	MPI_UNIVERSE_SIZE, 376, 397, 398, 673
33	MPI_UNSIGNED, 26, 176, 567, 667
34	MPI_UNSIGNED_CHAR, 26, 176, 178,
35	179, 667 MDI UNSIGNED LONG 26, 176, 567
36	MPI_UNSIGNED_LONG, 26, 176, 567,
37	667 MDI UNSIGNED LONG LONG 26, 176
38	MPI_UNSIGNED_LONG_LONG, 26, 176,
39	567, 667, 797
40	MPI_UNSIGNED_SHORT, 26, 176, 667
40	MPI_UNWEIGHTED, 16, 297, 298, 300,
42	301, 308, 310, 626, 675, 790, 796
43	MPI_VAL, 12, 648
43	MPI_VERSION, 336, 676
44 45	MPI_WCHAR, 26, 178, 179, 284, 536, 667,
45 46	797 MDI WEICHTS EMDTY 207 208 200
40 47	MPI_WEIGHTS_EMPTY, 297, 298, 300,
48	626, 675, 790 MDI WIN BASE 415, 650, 672
10	MPI_WIN_BASE, 415, 659, 673

MPI_WIN_CREATE_FLAVOR, 415, 673
MPI_WIN_DISP_UNIT, 415, 673
MPI_WIN_DUP_FN, <u>273</u> , 672
MPI_WIN_FLAVOR_ALLOCATE, 416,
673
MPI_WIN_FLAVOR_CREATE, 416, 673
MPI_WIN_FLAVOR_DYNAMIC, 416,
673
MPI_WIN_FLAVOR_SHARED, 416, 673
MPI_WIN_MODEL, 415, 437, 673
MPI_WIN_NULL, 414, 671
MPI_WIN_NULL_COPY_FN, <u>273</u> , 672
MPI_WIN_NULL_DELETE_FN, <u>273</u> , 672
MPI_WIN_SEPARATE, 416, 436, 437,
455,673
MPI_WIN_SIZE, 415, 673
MPI_WIN_UNIFIED, 416, 437, 456, 464,
673
MPI_WTIME_IS_GLOBAL, 337, 338,
357,655,670

#### **MPI Declarations Index**

This index refers to declarations needed in C, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI\_Aint, 16, <u>16</u>, 17, 19, 25, 85, <u>85</u>, 88, 90, 93, 102, 107, 108, 118, 139, 140, 405, 407, 408, 412, 419, 421, 425, 427, 429-434, 536, 539, 627, 656, 678MPI\_Comm, 12, 24, 230, 235–240, 242,  $244, 247, 248, \underline{249}, 259-262, 268,$ 271, 272, 669, 671, 678, 679 MPI\_Count, 17, 25, 678, 790, 791 MPI\_Datatype, <u>85</u>, 635, 667–669, 671, 678, 679 MPI\_Errhandler, <u>343</u>, 344–348, 649, 666, 671, 678, 679 MPI\_F08\_status, 651, 676, 678, 793 MPI\_File, 347, 348, 356, 493, 495, 497-499, 501, 503, 506, 510-519, 521-525, 527-532, 536, 544, 545, 649, 671, 678, 679 MPI\_Fint, 648, 676, 678, 797 MPI\_Group, 228, 228, 229–235, 240, 260, 416, 442, 443, 499, 649, 671, 678, 679 MPI\_Info, 339, 367, <u>367</u>, 368–371, 376, 379, 382, 388, 390–394, 417, 493, 496, 501, 503, 649, 669, 671, 678, 679,800 MPI\_Message, 649, 665, 671, 678, 791 MPI\_Offset, 16, 16, 17, 19, 25, 497, 498, 503, 506, 510-513, 518, 519, 525, 527, 528, 540, 547, 547, 647, 678 MPI\_Op, 174, 183, 185, 187, 189, 190, 192-195, 209-214, 425, 427, 429, 433, 434, 649, 670, 671, 678, 679 MPI\_Request, 49-51, 53, 54, 55, 57-61, 64, 71, 74–77, 476, 479, 512, 513, 517,

13 518, 522, 628, 649, 671, 678, 679 14MPI\_Status, <u>28</u>, 30–32, 53, 54, 57–61, 64, 1565, 68, 69, 70, 72, 79, 80, 113, 114, 16477, 482, 483, 510-512, 514-516, 17521, 523, 524, 528-532, 603, 18 650-652, 675, 678, 679, 790, 793 19MPI\_T\_cvar\_handle, 572, 572, 573, 574, 20676 21MPI\_T\_enum, 567, 567, 568, 570, 578, 676 22MPI\_T\_pvar\_handle, 580, 580, 581-583, 23676  $^{24}$ MPI\_T\_pvar\_session, 579, <u>579</u>, 580–583, 25676 26MPI\_Win, 273, 275, 285, 346, 355, 405, 27407, 408, 412, 414, 416, 417, 419, 28421, 425, 427, 429–434, 440, 29442-444, 446, 447, 449, 450, 649, 30 671, 678, 679 3132 33 34 3536 37 38 39 40 41 4243 4445464748

10 11

# 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. Fortran example prototypes are given near the text of the C name.

15MPI\_Comm\_copy\_attr\_function, 18, 19, 16268, 602, 672, 679 17MPI\_Comm\_delete\_attr\_function, 18, 268, 18 672, 679 19MPI\_Comm\_errhandler\_fn, 596, 796 20MPI\_Comm\_errhandler\_function, 18, 344, 21596, 598, 679, 796 22MPI\_Copy\_function, 18, 593, 673, 684 23MPI\_Datarep\_conversion\_function, 540,  $^{24}$ 672,680 25MPI\_Datarep\_extent\_function, 539, 680 26MPI\_Delete\_function, 18, 594, 673, 684 27MPI\_File\_errhandler\_fn, 596, 796 28MPI\_File\_errhandler\_function, 347, 596, 29679.796 30 MPI\_Grequest\_cancel\_function, 478, 680 31MPI\_Grequest\_free\_function, 477, 680 32 MPI\_Grequest\_query\_function, 477, 679 33 MPI\_Handler\_function, 18, 598, 790 34MPI\_Type\_copy\_attr\_function, 277, 672, 35 679 36 MPI\_Type\_delete\_attr\_function, 277, 672, 37 679, 794 38MPI\_User\_function, 183, 187, 679 39MPI\_Win\_copy\_attr\_function, 273, 672, 40679  $^{41}$ MPI\_Win\_delete\_attr\_function, 273, 672, 42679 43MPI\_Win\_errhandler\_fn, 596, 796  $^{44}$ MPI\_Win\_errhandler\_function, 346, 596, 45679, 796 464748

5 6

12

13

## **MPI** Function Index

The underlined page numbers refer to the function definitions.

MPI\_ABORT, 184, 342, 359, <u>362</u>, 400, 567.648.800 MPI\_ACCUMULATE, 403, 418, 425, 426, 428, 434, 437, 461, 468, 469, 797, 800 MPI\_ADD\_ERROR\_CLASS, 352, 353 MPI\_ADD\_ERROR\_CODE, 353 MPI\_ADD\_ERROR\_STRING, 354, 354 MPI\_ADDRESS, 18, 103, 597, 609, 611, 790 MPI\_ALLGATHER, 141, 145, 146, 165, 165, 166-168, 204 MPI\_ALLGATHERV, 141, 145, 146, 166, 167, 205 MPI\_ALLOC\_MEM, 339, 340, 341, 350, 406-409, 411, 413, 420, 448, 609, 612-614, 626, 794 MPI\_ALLOC\_MEM\_CPTR, 340, 609 MPI\_ALLREDUCE, 141, 144–146, 176, 183, 187, 188, 210, 798 MPI\_ALLTOALL, 141, 145, 146, 168, 168, 169-171, 206, 795 MPI\_ALLTOALLV, 141, 145, 146, 170, 170, 171, 173, 207, 795 MPI\_ALLTOALLW, 141, 145, 146, 172, 173, 209, 795 MPI\_ATTR\_DELETE, 18, 279, 594, 595 MPI\_ATTR\_GET, 18, 279, 595, 655, 656 MPI\_ATTR\_PUT, 18, 279, <u>595</u>, 655, 656, 658, 659 MPI\_BARRIER, 141, 145, 147, <u>147</u>, 148, 199, 458-460, 549 MPI\_BCAST, 141, 145, 148, 148, 149, 175, 199, 218 MPI\_BSEND, 38, 46 MPI\_BSEND\_INIT, 74, 77 MPI\_BUFFER\_ATTACH, 44, 53

MPI\_BUFFER\_DETACH, 45, 793 12MPI\_CANCEL, 41, 53, 64, 71, 71, 72, 73, 13 197. 361. 431. 475. 479 14MPI\_CART\_COORDS, 291, 305, 306, 799 15MPI\_CART\_CREATE, 258, 290-292, 292, 16293, 294, 304, 312, 313, 315, 627, 17798 18 MPI\_CART\_GET, 291, 304, 304, 799 19MPI\_CART\_MAP, 291, 313, 313, 792 20MPI\_CART\_RANK, 291, 305, 305, 799 21MPI\_CART\_SHIFT, 291, 310, 310, 311, 22314, 799 23MPI\_CART\_SUB, 291, 312, <u>312</u>, 313, 799  $^{24}$ MPI\_CARTDIM\_GET, 291, 304, 304, 799 25MPI\_CLOSE\_PORT, 389, 389, 393 26MPI\_COMM\_ACCEPT, 388, 390, 390, 27391, 398, 399 28MPI\_COMM\_C2F, 648 29MPI\_COMM\_CALL\_ERRHANDLER, 30 355, 356 31MPI\_COMM\_COMPARE, 237, 259 32 MPI\_COMM\_CONNECT, 350, 391, 391, 33 398, 399 34 MPI\_COMM\_CREATE, 235, 237, 240, 35 240, 241-245, 291, 796 36 MPI\_COMM\_CREATE\_ERRHANDLER, 37 18, 343, <u>343</u>, 345, 597, 681, 683, 38794 39 MPI\_COMM\_CREATE\_GROUP, 237, 40 242, 243, 244, 791 41 MPI\_COMM\_CREATE\_KEYVAL, 18, 42266, 267, 269, 270, 279, 593, 654, 43655, 680, 682, 794, 798  $^{44}$ MPI\_COMM\_DELETE\_ATTR, 18, 266, 45269-271, 272, 279, 595 46MPI\_COMM\_DISCONNECT, 279, 381, 47399, 400, 400 48

1 2 3

9

1	MPI_COMM_DUP, 230, 235, 237, 238,
2	238, 239, 241, 248, 249, 260, 262,
3	$\overline{266}, 269, 272, 279, 286, 593, 791$
4	MPI_COMM_DUP_FN, 18, 269, <u>269</u> , 270,
5	609, 672, 794
6	MPI_COMM_DUP_WITH_INFO, 237,
7	<u>238,</u> 239, 248, 791
8	MPI_COMM_F2C, <u>648</u>
9	MPI_COMM_FREE, 235, 238, 248, <u>248</u> ,
10	261, 262, 269, 270, 272, 279, 363,
11	381, 399, 400, 594
12	MPI_COMM_FREE_KEYVAL, 18, 266,
13	$\underline{270}, 279, 594$
14	MPI_COMM_GET_ATTR, 18, 266, 271,
15	$\underline{271}, 279, 336, 595, 656, 658$
16	MPI_COMM_GET_ERRHANDLER, 18,
17	$343, \underline{345}, 597, 799$
18	MPI_COMM_GET_INFO, <u>249</u> , 250, 791
19	MPI_COMM_GET_NAME, 282, <u>282</u> , 283,
20	798
21	MPI_COMM_GET_PARENT, 283, 377,
22	$\underline{380}, 381$
23	MPI_COMM_GROUP, 14, 228, 230, <u>230</u> ,
24	235, 236, 259, 343, 799
25	MPI_COMM_IDUP, 235, 237, 239, <u>239</u> ,
26	248, 249, 266, 269, 272, 279, 791
27	MPI_COMM_JOIN, 401, <u>401</u> , 402
28	MPI_COMM_KEYVAL_CREATE, 609
29	MPI_COMM_NULL_COPY_FN, 18, 269,
30	269, 270, 602, 654, 672, 794
31	MPI_COMM_NULL_DELETE_FN, 18,
32	269, 269, 270, 672
33	MPI_COMM_RANK, 236, <u>236</u> , 259, 609
34	MPI_COMM_REMOTE_GROUP, <u>260</u>
35	MPI_COMM_REMOTE_SIZE, 260, <u>260</u>
36	MPI_COMM_SET_ATTR, 18, 266, 269,
37	<u>271, 279, 594, 656, 659</u>
38	MPI_COMM_SET_ERRHANDLER, 18,
39	343, <u>344</u> , 597
40	MPI_COMM_SET_INFO, 248, 249, <u>249</u> ,
41 42	791
42 43	MPI_COMM_SET_NAME, <u>281</u> , 282
43 44	MPI_COMM_SIZE, <u>235</u> , 236, 259
44 45	MPI_COMM_SPAWN, 358, 364, 365,
45 46	$374-376, \underline{376}, 377, 379-381,$
40	383–385, 397–399 MDL COMM SDAWN MULTIDLE 250
48	MPI_COMM_SPAWN_MULTIPLE, 359, 265_274_275_281_282_282_208
	$365, 374, 375, 381, \underline{382}, 383, 398,$

399

- MPI\_COMM\_SPLIT, 237, 240, 241, 244, <u>244</u>, 245, 246, 286, 291, 292, 294, 312-314, 796
- MPI\_COMM\_SPLIT\_TYPE, <u>247</u>, 248, 792
- MPI\_COMM\_TEST\_INTER, 258, 259
- MPI\_COMM\_WORLD, 489
- MPI\_COMPARE\_AND\_SWAP, 403, 418, 430, 468
- MPI\_CONVERSION\_FN\_NULL, 541, 672
- MPI\_DIMS\_CREATE, 291, 293, 293
- MPI\_DIST\_GRAPH\_CREATE, 248, 290, 291, 296, <u>298</u>, 299, 301, 310, 315, 796
- MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, 248, 290, 291, 296, <u>296</u>, 297, 301, 310, 314, 792, 796
- MPI\_DIST\_GRAPH\_NEIGHBOR\_COUNT, 310
- MPI\_DIST\_GRAPH\_NEIGHBORS, 291, 308, <u>309</u>, 310, 315, 792, 796
- MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT, 291, 308, <u>308</u>, 309, 789, 796
- MPI\_DUP\_FN, 18, 269, 594, 673
- MPI\_ERRHANDLER\_C2F, 649
- MPI\_ERRHANDLER\_CREATE, 18, 597, 790, 794
- MPI\_ERRHANDLER\_F2C, 649
- MPI\_ERRHANDLER\_FREE, 343, 348, 799
- MPI\_ERRHANDLER\_GET, 18, 597, 790, 799
- MPI\_ERRHANDLER\_SET, 18, 597, 790
- MPI\_ERROR\_CLASS, 349, 352, 352, 590
- MPI\_ERROR\_STRING, 349, 349, 352, 354
- MPI\_EXSCAN, 142, 145, 176, 194, <u>194</u>, 214, 796
- MPI\_F\_SYNC\_REG, 103, 600, 616, 617, 617, 636, 639, 640, 642, 794
- MPI\_FETCH\_AND\_OP, 403, 418, 426, 428, 429
- MPI\_FILE\_C2F, 649
- MPI\_FILE\_CALL\_ERRHANDLER, 356, 356
- MPI\_FILE\_CLOSE, 400, 493, 494, 495, 496
- MPI\_FILE\_CREATE\_ERRHANDLER, 343, <u>347</u>, 348, 681, 684, 794

MPI_FILE_DELETE, 495, 496, <u>496</u> , 500,
503, 552
MPI_FILE_F2C, $\underline{649}$
$MPI\_FILE\_GET\_AMODE, 499, \underline{499}$
MPI_FILE_GET_ATOMICITY, 545, 545
MPI_FILE_GET_BYTE_OFFSET, 514,
519, <u>519</u> , 520, 526
MPI_FILE_GET_ERRHANDLER, 343,
348, 552, 799
MPI_FILE_GET_GROUP, 499, <u>499</u>
MPI_FILE_GET_INFO, 501, <u>501</u> , 503, 800
MPI_FILE_GET_POSITION, 519, 519
MPI_FILE_GET_POSITION_SHARED,
524, 525, <u>525</u> , 545
MPI_FILE_GET_SIZE, <u>498</u> , 499, 548
MPI_FILE_GET_TYPE_EXTENT, 535,
536, 542
MPI_FILE_GET_VIEW, 506, <u>506</u>
MPI_FILE_IREAD, 507, 517, 517, 526,
543
MPI_FILE_IREAD_AT, 507, <u>512</u> , 513
MPI_FILE_IREAD_A1, 507, 512, 513 MPI_FILE_IREAD_SHARED, 507, 522,
$\frac{522}{100}$
MPI_FILE_IWRITE, 507, 518, <u>518</u>
MPI_FILE_IWRITE_AT, 507, 513, <u>513</u>
MPI_FILE_IWRITE_SHARED, 507, <u>522</u> ,
523
MPI_FILE_OPEN, 351, 486, 493, <u>493</u> ,
494, 495, 500, 502–504, 520, 547,
552, 553
MPI_FILE_PREALLOCATE, 497, 498,
$\underline{498}, 543, 547, 548$
MPI_FILE_READ, 506, 507, 514, <u>514</u> ,
515,517,547,548
MPI_FILE_READ_ALL, 507, 515, <u>515</u> ,
526, 527
MPI_FILE_READ_ALL_BEGIN, 507,
$526, 527, \underline{529}, 543, 642$
MPI_FILE_READ_ALL_END, 507, 526,
527, 529, 543, 642
MPI_FILE_READ_AT, 507, 510, <u>510</u> , 511,
513
MPI_FILE_READ_AT_ALL, 507, 510,
511
MPI_FILE_READ_AT_ALL_BEGIN, 507,
527, 642
MPI_FILE_READ_AT_ALL_END, 507,
<u>528</u> , 642

$MPI\_FILE\_READ\_ORDERED, 507, \underline{523},$	1
524	2
MPI_FILE_READ_ORDERED_BEGIN,	3 4
$507, \underline{531}, 642$	5
MPI_FILE_READ_ORDERED_END, 507, <u>531</u> , 642	6
MPI_FILE_READ_SHARED, 507, 521,	7
<u>521</u> , 522, 524	8
MPI_FILE_SEEK, <u>518</u> , 519	9
MPI_FILE_SEEK_SHARED, 524, 525,	10
525, 526, 545	11
MPI_FILE_SET_ATOMICITY, 495, 544,	12
544	13
MPI_FILE_SET_ERRHANDLER, 343,	14
348, 552	15
MPI_FILE_SET_INFO, 500, 501, <u>501</u> ,	16
502, 503, 800	17
MPI_FILE_SET_SIZE, 497, <u>497</u> , 498, 543,	18
546-548	19
MPI_FILE_SET_VIEW, 97, 351, 494, 500,	20
$\begin{array}{c} \text{MI1} 1\_\text{F1LE}\_\text{SE1}\_\text{V1EW}, 97, 351, 494, 500, \\ 502, 503, 503, 504, 505, 519, 526, \end{array}$	21
, ,, , , , , ,	22
533, 539, 547, 553, 800, 801	22
MPI_FILE_SYNC, 496, 507, 543–545, <u>545</u> , 550	20
	25
MPI_FILE_WRITE, 506, 507, 516, <u>516</u> , 517, 518, 547	26
MPI_FILE_WRITE_ALL, 507, <u>516</u> , 517	27
MPI_FILE_WRITE_ALL_BEGIN, 507,	28
530, 628, 642	29
MPI_FILE_WRITE_ALL_END, 507, <u>530</u> ,	30
$\begin{array}{c} \text{MITE_ALL_END, 507, } \underline{550}, \\ 642 \end{array}$	31
MPI_FILE_WRITE_AT, 507, 511, <u>511</u> ,	32
$\frac{MF1_FILE_WRITE_AT, 507, 511, 511}{512, 513}$	33
,	34
MPI_FILE_WRITE_AT_ALL, 507, 512,	35
$\frac{512}{100000000000000000000000000000000000$	36
MPI_FILE_WRITE_AT_ALL_BEGIN,	37
507, <u>528</u> , 607, 642	38
MPI_FILE_WRITE_AT_ALL_END, 507,	39
<u>529</u> , 642	
MPI_FILE_WRITE_ORDERED, 507,	40
523, 524, <u>524</u>	41
MPI_FILE_WRITE_ORDERED_BEGIN,	42
$507, \underline{532}, 642$	43
MPI_FILE_WRITE_ORDERED_END,	44
$507, \underline{532}, 642$	45
$MPI\_FILE\_WRITE\_SHARED, 507, \underline{521},$	46
522 - 524	47
	48

1	MPI_FINALIZE, 15, 22, 336, 337, 359,
2	359, 360-364, 400, 485, 494, 563,
3	$\overline{572}$ , 584, 586, 648, 650, 651, 792,
4	800
5	MPI_FINALIZED, 358, 361, 363, <u>363</u> , 364,
6	490, 648
7	MPI_FREE_MEM, 340, <u>340</u> , 341, 350,
8	407,  409,  609
9	MPI_GATHER, 141, 144–146, <u>149</u> , 151,
10	152,159,160,165,175,201
11	MPI_GATHERV, 141, 145, 146, 151, <u>151</u> ,
12	152, 153, 161, 167, 202
13	MPI_GET, 403, 418, <u>421</u> , 422, 428, 433,
14	437, 458, 460, 461, 639, 800
15	MPI_GET_ACCUMULATE, 403, 418,
16	$426, \underline{427}, 428, 435, 461, 468$
17	MPI_GET_ADDRESS, 18, 85, 102, <u>102</u> ,
18	103, 115, 412, 597, 609, 612, 630,
19	635, 653, 654
20	MPI_GET_COUNT, 31, <u>31</u> , 32, 52, 114,
21	115, 431, 483, 509, 791, 797
22	MPI_GET_ELEMENTS, 52, 113, <u>113</u> ,
23 24	114, 115, 483, 484, 509, 791
24 25	MPI_GET_ELEMENTS_X, 52, 113, 114,
26	<u>114</u> , 115, 483, 509, 790, 791 MPI_GET_LIBRARY_VERSION, <b>336</b> ,
27	<u>336</u> , 358, 361, 790
28	MPI_GET_PROCESSOR_NAME, 338,
29	<u>338</u> , 339, 799
30	MPI_GET_VERSION, <u>335</u> , 336, 358, 361,
31	490, 615
32	MPI_GRAPH_CREATE, 290, 291, 294,
33	<u>294</u> , 296, 300, 303, 307, 314, 315,
34	798, 799
35	MPI_GRAPH_GET, 291, 303, 303
36	MPI_GRAPH_MAP, 291, 314, <u>314</u>
37	MPI_GRAPH_NEIGHBORS, 291, 306,
38	306, 307, 315, 796
39	MPI_GRAPH_NEIGHBORS_COUNT,
40	$291, 306, \underline{306}, 307, 796$
41	MPI_GRAPHDIMS_GET, 291, 303, <u>303</u>
42	MPI_GREQUEST_COMPLETE, 476–479,
43	$\underline{479},  480$
44	MPI_GREQUEST_START, <u>476</u> , 477, 681,
45	684, 797
46	MPI_GROUP_C2F, $\underline{649}$
47	MPI_GROUP_COMPARE, <u>229</u> , 232
48	MPI_GROUP_DIFFERENCE, <u>231</u>

MPI\_GROUP\_EXCL, <u>232</u>, 233, 234 MPI\_GROUP\_F2C, 649 MPI\_GROUP\_FREE, 235, 235, 236, 343, 799 MPI\_GROUP\_INCL, 232, 232, 234 MPI\_GROUP\_INTERSECTION, 231 MPI\_GROUP\_RANGE\_EXCL, 234, 234 MPI\_GROUP\_RANGE\_INCL, 233, 234 MPI\_GROUP\_RANK, 228, 236 MPI\_GROUP\_SIZE, 228, 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, 210 MPI\_IALLTOALL, 141, 145, 146, 206 MPI\_IALLTOALLV, 141, 145, 146, 207 MPI\_IALLTOALLW, 141, 145, 146, 208 MPI\_IBARRIER, 141, 145, 197, 198, 199, 218MPI\_IBCAST, 141, 145, 199, 200, 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, 486, 791 MPI\_IMRECV, 67-69, 71, 71, 791 MPI\_INEIGHBOR\_ALLGATHER, 292, 325, 792 MPI\_INEIGHBOR\_ALLGATHERV, 292, 326, 792 MPI\_INEIGHBOR\_ALLTOALL, 292, 327, 792MPI\_INEIGHBOR\_ALLTOALLV, 292, 328, 792 MPI\_INEIGHBOR\_ALLTOALLW, 292, 329, 792 MPI\_INFO\_C2F, 649 MPI\_INFO\_CREATE, 368, 368 MPI\_INFO\_DELETE, 350, 369, 369, 371 MPI\_INFO\_DUP, 371, 371 MPI\_INFO\_F2C, 649 MPI\_INFO\_FREE, 250, <u>371</u>, 418, 501 MPI\_INFO\_GET, 367, 369, 800 MPI\_INFO\_GET\_NKEYS, 367, 370, 370, 371,800

MPI_INFO_GET_NTHKEY, <u>367</u> , <u>371</u> ,
800
MPI_INFO_GET_VALUELEN, <u>367</u> , <u>370</u> , 800
MPI_INFO_SET, 368, <u>368</u> , 369, 371
MPI_INIT, 15, 22, 227, 336, 337, 357, <u>357</u> ,
358, 361 - 364, 377 - 379, 381, 397,
398,  487,  488,  490,  559,  563,  566,
567, 572, 584, 590, 647, 648, 650,
651, 792, 795, 796
MPI_INIT_THREAD, 227, 358, 363, <u>487</u> ,
488-490, 566, 590, 647, 792, 796
MPI_INITIALIZED, 358, 361, 362, <u>362</u> ,
363, 364, 490, 648
MPI_INTERCOMM_CREATE, 237, 243,
260, 261, <u>261</u> , 262, 792
MPI_INTERCOMM_MERGE, 237, 243,
$258, 260-262, \underline{262}, 794$
MPI_IPROBE, 31, 64, <u>64</u> , 65–69, 71, 486,
791
MPI_IRECV, <u>51</u> , 71, 629, 630, 633, 634
MPI_IREDUCE, 141, 145, 146, <u>209</u> , 210
MPI_IREDUCE_SCATTER, 141, 145,
146, <u>212</u>
MPI_IREDUCE_SCATTER_BLOCK,
141, 145, 146, <u>211</u>
MPI_IRSEND, <u>51</u>
,
MPI_IS_THREAD_MAIN, 488, <u>490</u>
MPI_ISCAN, 142, 145, <u>213</u>
MPI_ISCATTER, 141, 145, 146, <u>202</u>
MPI_ISCATTERV, 141, 145, 146, <u>203</u>
MPI_ISEND, 11, <u>49</u> , 77, 609–611, 628,
629,634,635
MPI_ISSEND, <u>50</u>
MPI_KEYVAL_CREATE, 18, <u>593</u> , 595,
685
MPI_KEYVAL_FREE, 18, 279, <u>594</u>
MPI_LOCK_ALL, 405
MPI_LOOKUP_NAME, 350, 388, 392,
394, <u>394</u>
MPI_MESSAGE_C2F, <u>649</u> , 791
MPI_MESSAGE_F2C, <u>649</u> , 791
MPI_MPROBE, 64, 67–69, <u>69</u> , 71, 486,
791
MPI_MRECV, 67–70, <u>70</u> , 71, 791
MPI_NEIGHBOR_ALLGATHER, 291,
$\underline{316}, 317, 319, 325, 792$

MPI_NEIGHBOR_ALLGATHERV, 291,	1
$\underline{318}, 326, 792$	2
MPI_NEIGHBOR_ALLTOALL, 291, <u>319</u> ,	3
320, 327, 792	4
MPI_NEIGHBOR_ALLTOALLV, 291,	5
321, 328, 792	6
MPI_NEIGHBOR_ALLTOALLW, 292,	7
322, 323, 330, 792	8
MPI_NULL_COPY_FN, 18, 19, 269, <u>594</u> ,	9
673	10
MPI_NULL_DELETE_FN, 18, 269, <u>594</u> ,	11
673	12
MPI_OP_C2F, <u>649</u>	13
MPI_OP_COMMUTATIVE, <u>189</u> , 796	14
MPI_OP_CREATE, 183, <u>183</u> , 185, 609,	15
680, 682, 793, 794	16
MPI_OP_F2C, <u>649</u>	17
MPI_OP_FREE, <u>185</u>	18
MPI_OPEN_PORT, 388, <u>388</u> , 390, 391,	19
393, 394	20
MPI_PACK, 46, <u>132</u> , 135, 138, 537, 540	21
MPI_PACK_EXTERNAL, 7, 138, <u>139</u> ,	22
622, 798	23
MPI_PACK_EXTERNAL_SIZE, <u>140</u>	24
MPI_PACK_SIZE, 46, 135, <u>135</u> , 791	25
MPI_PCONTROL, 558, 559, <u>559</u>	26
MPI_PROBE, 29, 31, 32, 64, 65, <u>65</u> , 66,	27
67, 69, 71, 486, 791	28
MPI_PUBLISH_NAME, 388, 392, <u>392</u> ,	29
393, 394	30
MPI_PUT, 403, 418, <u>419</u> , 421, 425, 426,	31
$\begin{array}{c} 432, 437, 443, 452, 454, 455, 459, \\ \end{array}$	32
452, 457, 445, 452, 454, 455, 459, 460, 628, 639, 800	33
400, 028, 039, 800 MPI_QUERY_THREAD, <u>489</u> , 490	34
	35
MPI_RACCUMULATE, 403, 418, 426,	36
428, <u>433</u> , 434 MDI DECV 24, 28, 20, 22, 65, 67, 68, 84	37
MPI_RECV, 24, <u>28</u> , 30–32, 65, 67, 68, 84,	38
112, 113, 133, 134, 142, 150, 219,	39
484, 549, 584, 635, 639, 640	40
MPI_RECV_INIT, <u>76</u> , 77	40
MPI_REDUCE, 141, 145, 146, 174, <u>174</u> ,	42
175, 176, 183-185, 188, 191, 192, 104, 200, 210, 425, 426, 420, 420, 420, 420, 420, 420, 420, 420	
194, 209, 210, 425, 426, 428, 429,	43
797	44
MPI_REDUCE_LOCAL, 175, 176, 183,	45
$\frac{189}{794}, 794, 796$	46
MPI_REDUCE_SCATTER, 141, 145, 146,	47
$176, 183, 191, 192, \underline{192}, 212$	48

1	MPI_REDUCE_SCATTER_BLOCK, 141,
2	$145, 146, 176, 183, 190, \underline{190}, 191,$
3	211, 796
4	MPI_REGISTER_DATAREP, 351, <u>537</u> ,
5	539-541, 553, 609, 682, 684
6	MPI_REQUEST_C2F, $\underline{649}$
7	MPI_REQUEST_F2C, $\underline{649}$
8	MPI_REQUEST_FREE, 55, <u>55</u> , 72, 78,
9	197,  359,  431,  478,  479,  795
10	MPI_REQUEST_GET_STATUS, 32, 64,
11	$\underline{64}, 477, 795$
12	MPI_RGET, 403, 418, <u>432</u> , 433
13	MPI_RGET_ACCUMULATE, 403, 418,
14	$426, 428, \underline{434}, 435$
15	MPI_RPUT, 403, 418, <u>431</u> , 432
16	MPI_RSEND, <u>39</u>
17	MPI_RSEND_INIT, <u>76</u>
18 19	MPI_SCAN, 142, 145, 176, 183, 193, <u>193</u> ,
20	195, 213
20	MPI_SCATTER, 141, 145, 146, 159, <u>159</u> ,
21	161, 162, 191, 203 MPI_SCATTERV, 141, 145, 146, 161, <u>161</u> ,
23	162, 192, 203
24	MPI_SEND, 23, <u>24</u> , 25, 32, 34, 84, 112,
25	$\begin{array}{c} 133, 219, 494, 549, 560, 607-610, \end{array}$
26	635, 636, 639
27	MPI_SEND_INIT, <u>74</u> , 77
28	MPI_SENDRECV, 79, 310
29	MPI_SENDRECV_REPLACE, 80
30	MPI_SIZEOF, 600, 623, <u>623</u>
31	MPI_SSEND, $\underline{39}$
32	MPI_SSEND_INIT, $\underline{75}$
33	MPI_START, 77, <u>77</u> , 78
34	MPI_STARTALL, <u>77</u> , 78
35	$MPI\_STATUS\_C2F, \underline{650}$
36	MPI_STATUS_C2F08, <u>651</u> , 793
37	MPI_STATUS_F082C, <u>651</u> , 793
38	MPI_STATUS_F082F, <u>652</u> , 793
39	MPI_STATUS_F2C, <u>650</u>
40	MPI_STATUS_F2F08, <u>652</u> , 793
41	MPI_STATUS_SET_CANCELLED, <u>483</u>
42	MPI_STATUS_SET_ELEMENTS, <u>482</u> ,
43 44	483 MDI CTATUS SET ELEMENTS X 492
44	MPI_STATUS_SET_ELEMENTS_X, 483,
45	483, 790 MPI_T_CATEGORY_CHANGED, 589
47	MPI_T_CATEGORY_GET_CATEGORIES,
48	589, <u>589</u> , 590, 591
	009, 009, 000, 001

MPI_T_CATEGORY_GET_CVARS, 588,
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, 589,
589, 590, 591
$MPI\_T\_CVAR\_GET\_INFO, 567, 569,$
$570,  \underline{570},  571,  573,  574,  590,  591$
MPI_T_CVAR_GET_NUM, <u>569</u> , 573
$MPI_T_CVAR_HANDLE\_ALLOC, \underline{572},$
573, 574, 591
MPI_T_CVAR_HANDLE_FREE, 573,
573, 591
$MPI\_T\_CVAR\_READ, \frac{573}{2}$
$MPI_T_CVAR_WRITE, \frac{574}{2}$
MPI_T_ENUM_GET_INFO, 568, <u>568</u> , 591
$MPI_T\_ENUM\_GET\_ITEM, 568, \underline{568},$
591
MPI_T_FINALIZE, 566, <u>566</u> , 567
MPI_T_INIT_THREAD, <u>565</u> , 566, 567
MPI_T_PVAR_GET_INFO, 567, 577, 578,
578, 580-584, 590, 591
MPI_T_PVAR_GET_NUM, <u>578</u> , 581
MPI_T_PVAR_HANDLE_ALLOC, 580,
$\underline{580}, 582, 583, 591$
$MPI_T_PVAR_HANDLE_FREE, 581,$
581, 591
MPI_T_PVAR_READ, 582, <u>582</u> , 583, 591
MPI_T_PVAR_READRESET, 579, 583,
583, 591
MPI_T_PVAR_RESET, 583, <u>583</u> , 591
$MPI_T_PVAR\_SESSION\_CREATE, \frac{579}{2},$
591
$MPI_T_PVAR\_SESSION\_FREE, \underline{580}, 591$
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, 52–54, <u>54</u> , 55, 57, 58,
72, 78, 359, 479, 508, 509
MPI_TEST_CANCELLED, 52–54, <u>72</u> , 73,
477,  484,  509
MPI_TESTALL, 57, 60, <u>60</u> , 477–479, 482,
485
MPI_TESTANY, 53, 57, 58, <u>58</u> , 62,
477-479, 482, 485
MPI_TESTSOME, 57, <u>61</u> , 62, 477–479,
482, 485

Unofficial Draft for Comment Only

MPI_TOPO_TEST, 291, 302, <u>302</u>
MPI_TYPE_C2F, <u>648</u>
MPI_TYPE_COMMIT, 110, <u>110</u> , 649
MPI_TYPE_CONTIGUOUS, 12, 85, <u>85</u> ,
87,105,118,492,535
MPI_TYPE_CREATE_DARRAY, 12, 31,
$98, \underline{98}, 118$
MPI_TYPE_CREATE_F90_COMPLEX,
$12, 118, 119, 177, 537, 600, \underline{619},$
621, 622
MPI_TYPE_CREATE_F90_INTEGER,
$12, 118, 119, 176, 537, 600, \underline{620},$
621, 622
MPI_TYPE_CREATE_F90_REAL, 12,
$118, 119, 177, 537, 600, 619, \underline{619}, \\620-622, 795$
MPI_TYPE_CREATE_HINDEXED, 12,
18, 85, 90, <u>90</u> , 93, 94, 118, 597
MPI_TYPE_CREATE_HINDEXED_BLOCK,
12, 85, 92, <u>92</u> , 118, 791
MPI_TYPE_CREATE_HVECTOR, 12,
$18, 85, 88, \underline{88}, 118, 597$
MPI_TYPE_CREATE_INDEXED_BLOCK,
12, 92, 92, 118
MPI_TYPE_CREATE_KEYVAL, 266,
276, 279, 655, 681, 683, 798
MPI_TYPE_CREATE_RESIZED, 18, 85,
$105, 108, \underline{108}, 118, 535, 598, 793$
MPI_TYPE_CREATE_STRUCT, 12, 18,
$85, 93, \underline{93}, 94, 105, 118, 173, 597$
MPI_TYPE_CREATE_SUBARRAY, 12,
$14, \underline{95}, 97, 99, 118$
MPI_TYPE_DELETE_ATTR, 266, 279,
<u>279,</u> 793
MPI_TYPE_DUP, 12, 111, <u>111</u> , 118, 793
MPI_TYPE_DUP_FN, 276, <u>276</u> , 672 MPI_TYPE_EXTENT, 18, 597, 790
MPI_1 YPE_EXTEN1, 18, 597, 790 MPI_TYPE_F2C, 648
MPI_TYPE_FREE, <u>111</u> , 119, 277
$MPI_TYPE_FREE_KEYVAL, 266, 277,$
279
MPI_TYPE_GET_ATTR, 266, <u>278</u> , 279,
<u>656, 793</u>
MPI_TYPE_GET_CONTENTS, 117, <u>118</u> ,
119–121
MPI_TYPE_GET_ENVELOPE, <u>116</u> , 117,
119, 120, 621

MPI_TYPE_GET_EXTENT, 18, <u>107</u> ,	1
109,  597,  624,  653	2
MPI_TYPE_GET_EXTENT_X, <u>107</u> , 790	3
MPI_TYPE_GET_NAME, <u>284</u> , 794	4
MPI_TYPE_GET_TRUE_EXTENT, 108,	5
<u>108</u>	6
MPI_TYPE_GET_TRUE_EXTENT_X,	7
108, 109, 790	8
MPI_TYPE_HINDEXED, 18, 597, 790	9
MPI_TYPE_HVECTOR, 18, 597, 790	10
MPI_TYPE_INDEXED, 12, 89, <u>89</u> , 90, 91,	11
118	12
MPI_TYPE_LB, 18, 597, 790	13
MPI_TYPE_MATCH_SIZE, 600, 623,	14
<u>623</u> , 794	15
MPI_TYPE_NULL_COPY_FN, 276, <u>276</u> ,	16
672	17
MPI_TYPE_NULL_DELETE_FN, <u>276</u> ,	18
672, 794	19
MPI_TYPE_SET_ATTR, 266, <u>278</u> , 279,	20
656, 659, 793	21
MPI_TYPE_SET_NAME, <u>284</u> , 793	22
MPI_TYPE_SIZE, 104, <u>104</u> , 560, 791	23
MPI_TYPE_SIZE_X, 104, <u>104</u> , 790	24
MPI_TYPE_STRUCT, 18, 597, 790	25
MPI_TYPE_UB, 18, 597, 790	26
MPI_TYPE_VECTOR, 12, 86, 86, 87, 88,	27
90, 118	28
MPI_UNPACK, <u>133</u> , 134, 138, 540	29
MPI_UNPACK_EXTERNAL, 7, <u>139</u> , 622	30
	31
MPI_UNPUBLISH_NAME, 350, 393, <u>393</u>	32
MPI_WAIT, 30, 32, 52, 53, <u>53</u> , 54–56, 58,	33
59, 72, 78, 197, 219, 359, 475, 479,	34
486, 508, 509, 526, 543, 544, 628,	
634, 635, 639	35
MPI_WAITALL, 57, 59, <u>59</u> , 60, 197, 220,	36
431, 477–479, 482, 485, 486	37
MPI_WAITANY, 41, 53, 57, <u>57</u> , 58, 62,	38
477-479, 482, 485, 486	39
MPI_WAITSOME, 57, <u>60</u> , 61–63, 477–479,	40
482,  485,  486	41
MPI_WIN_ALLOC, 409	42
$MPI\_WIN\_ALLOCATE, 404, 407, \underline{407},$	43
415, 416, 420, 448, 609, 613, 614	44
MPI_WIN_ALLOCATE_CPTR, 408	45
MPI_WIN_ALLOCATE_SHARED, 404,	46
$\underline{408},409,411,416$	47
	48

1	MPI_WIN_ALLOCATE_SHARED_CPTR,
2	409
3	MPI_WIN_ATTACH, 411–413, <u>413</u> , 414,
4	448
5	MPI_WIN_C2F, <u>649</u>
6	MPI_WIN_CALL_ERRHANDLER, <u>355</u> ,
7	356
8	MPI_WIN_COMPLETE, 415, 438, 442,
9	443-445, 453, 454, 460
10	MPI_WIN_CREATE, 404, <u>404</u> , 406–409,
11	$412, 413, 415, 416, \overline{452}, 486$
12	MPI_WIN_CREATE_DYNAMIC, 351,
13	404, 411, 412, 412, 413, 415, 416,
14	453
15	MPI_WIN_CREATE_ERRHANDLER,
16	343, 345, 346, 681, 683, 794
17	MPI_WIN_CREATE_KEYVAL, 266, 273,
18	279, 655, 680, 683, 798
19	MPI_WIN_DELETE_ATTR, 266, 275,
20	279
21	MPI_WIN_DETACH, 411, 413, <u>414</u> , 415
22	MPI_WIN_DUP_FN, 273, <u>273</u> , 672
23	MPI_WIN_F2C, <u>649</u>
24	MPI_WIN_FENCE, 415, 422, 437, 440,
25	440, 441, 442, 451, 453, 454, 457,
26	$\frac{1}{462}, 639$
27	MPI_WIN_FLUSH, 410, 431, 432, 449,
28	449, 453, 454, 457, 468, 469
29	MPI_WIN_FLUSH_ALL, 431, 432, 449,
30	453, 454, 457
31	MPI_WIN_FLUSH_LOCAL, 431, 450, 453
32	MPI_WIN_FLUSH_LOCAL_ALL, 431,
33	450, 450, 453
34	MPI_WIN_FREE, 274, 400, 414, <u>414</u> , 415
35	MPI_WIN_FREE_KEYVAL, 266, 274, 279
36	MPI_WIN_GET_ATTR, 266, 275, 279,
37	415, 416, 656, 659
38	MPI_WIN_GET_ERRHANDLER, 343,
39	346, 799
40	MPI_WIN_GET_GROUP, 416, <u>416</u>
41	MPI_WIN_GET_INFO, 417, 417, 791
42	MPI_WIN_GET_NAME, 285
43	MPI_WIN_LOCK, 405, 438, 446, 447–449,
44	451, 454, 458, 459
45	MPI_WIN_LOCK_ALL, 438, <u>446</u> , 447,
46	451, 454, 459, 468
47	MPI_WIN_NULL_COPY_FN, 273, <u>273</u> ,
48	672

```
MPI_WIN_NULL_DELETE_FN, 273, 672
MPI_WIN_POST, 415, 438, 442, 443, 443,
       444-446, 448, 451, 454, 460, 462
MPI_WIN_SET_ATTR, 266, 275, 279,
       415, 656, 659
MPI_WIN_SET_ERRHANDLER, 343,
       346
MPI_WIN_SET_INFO, 417, 417, 791
MPI_WIN_SET_NAME, 285
MPI_WIN_SHARED_QUERY, 409, 410
MPI_WIN_SHARED_QUERY_CPTR,
       411
MPI_WIN_START, 438, 442, <u>442</u>, 443,
       445, 446, 451, 460, 467
MPI_WIN_SYNC, 450, 450, 454, 456, 460,
       468, 469
MPI_WIN_TEST, <u>444</u>, 445
MPI_WIN_UNLOCK, 432, 438, 447, 448,
       449, 453, 454, 457, 458
MPI_WIN_UNLOCK_ALL, 432, 438, 447,
       453, 454, 457, 468
MPI_WIN_WAIT, 415, 438, 444, 444, 445,
       448, 454, 457, 460, 461
MPI_WTICK, 19, 357, 357
MPI_WTIME, 19, 338, <u>356</u>, 357, 560, 577
MPI_xxx_FREE, 360
mpiexec, 358, 363, 364, <u>364</u>, 488
mpirun, 364
PMPI_, 557
```

```
PMPI_SEND, 608
PMPI_WTICK, 19
PMPI_WTIME, 19
```