MPI: A Message-Passing Interface Standard Version 3.2

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

May 8, 2019

Issue #96 + PR97: Semantic Terms update

See changes on pages i; xv; 11-13; 39-40; 50; 57; 67; 71-73; 218; 391; 554; 719-721; 835 for Chicago meeting May 28-31, 2019

1	This document describes the Message-Passing Interface (MPI) standard, version 3.1.
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 (May 5, 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 4, 2009) added additional
13	clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension
14	of MPI-2.2. This version, MPI-3.1, adds clarifications and minor extensions to MPI-3.0
15	
16	Comments . Please send comments on MPI to the MPI Forum as follows:
17	
18	1. Subscribe to http://lists.mpi-forum.org/mailman/listinfo.cgi/mpi-comments
19	
20	2. Send your comment to: mpi-comments@mpi-forum.org, together with the URL of
21	the version of the MPI standard and the page and line numbers on which you are
22	commenting. Only use the official versions.
23	Your comment will be forwarded to MPI Forum committee members for consideration.
24	Messages sent from an unsubscribed e-mail address will not be considered.
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, 2015 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 3.1: June 4, 2015. This document contains mostly corrections and clarifications to the MPI-3.0 document. The largest change is a correction to the Fortran bindings introduced in MPI-3.0. Additionally, new functions added include routines to manipulate MPI_Aint values in a portable manner, nonblocking collective I/O routines, and routines to get the index value by name for MPI_T performance and control variables.

Version 3.0: September 21, 2012. Coincident with the development of MPI-2.2, the MPI Forum began discussions of a major extension to MPI. This document contains the MPI-3 Standard. This draft version of the MPI-3 standard contains significant extensions to MPI functionality, including nonblocking collectives, new one-sided communication operations, and Fortran 2008 bindings. Unlike MPI-2.2, this standard is considered a major update to the MPI standard. As with previous versions, new features have been adopted only when there were compelling needs for the users. Some features, however, may have more than a minor impact on existing MPI implementations.

Version 2.2: September 4, 2009. This document contains mostly corrections and clarifications to the MPI-2.1 document. A few extensions have been added; however all correct MPI-2.1 programs are correct MPI-2.2 programs. New features were adopted only when there were compelling needs for users, open source implementations, and minor impact on existing MPI implementations.

Version 2.1: June 23, 2008. This document combines the previous documents MPI-1.3 (May 30, 2008) and MPI-2.0 (July 18, 1997). Certain parts of MPI-2.0, such as some sections of Chapter 4, Miscellany, and Chapter 7, Extended Collective Operations, have been merged into the Chapters of MPI-1.3. Additional errata and clarifications collected by the MPI Forum are also included in this document.

Version 1.3: May 30, 2008. This document combines the previous documents MPI-1.1 (June 12, 1995) and the MPI-1.2 Chapter in MPI-2 (July 18, 1997). Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 are also included in this document.

Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. A miscellany chapter discusses items that do not fit elsewhere, in particular language interoperability.

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the standard "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only new function in MPI-1.2 is one for identifying to which version of the MPI Standard the implementation conforms. There are small differences between MPI-1 and MPI-1.1. There are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2 and MPI-2.

 $45 \\ 46$

 $\mathbf{2}$

 24

Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum
 reconvened to correct errors and make clarifications in the MPI document of May 5, 1994,
 referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes
 from Version 1.0 are minor. A version of this document with all changes marked is available.

⁶ Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation
 ⁷ from over 40 organizations, has been meeting since January 1993 to discuss and define a set
 ⁸ of library interface standards for message passing. MPIF is not sanctioned or supported by
 ⁹ any official standards organization.

¹⁰ The goal of the Message-Passing Interface, simply stated, is to develop a widely used ¹¹ standard for writing message-passing programs. As such the interface should establish a ¹² practical, portable, efficient, and flexible standard for message-passing.

¹³ This is the final report, Version 1.0, of the Message-Passing Interface Forum. This ¹⁴ document contains all the technical features proposed for the interface. This copy of the ¹⁵ draft was processed by IAT_EX on May 5, 1994.

Contents

A	cknov	wledgments	ix			
1 Introduction to MPI						
	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	Background of MPI-3.1.	4			
	1.8	Who Should Use This Standard?	5			
	1.9	What Platforms Are Targets for Implementation?	5			
	1.10	What Is Included in the Standard?	5			
	1.11	What Is Not Included in the Standard?	6			
		Organization of This Document	6			
2		Terms and Conventions	9			
4	2.1	Document Notation	9 9			
	$\frac{2.1}{2.2}$		9 9			
		Naming Conventions	-			
	2.3	Procedure Specification	10 11			
	2.4	Semantic Terms				
	2.5	Data Types	14			
		2.5.1 Opaque Objects	14			
		2.5.2 Array Arguments	16			
		2.5.3 State	16			
		2.5.4 Named Constants	16			
		2.5.5 Choice	18			
		2.5.6 Absolute Addresses and Relative Address Displacements	18			
		2.5.7 File Offsets	18			
		2.5.8 Counts	18			
	2.6	Language Binding	19			
		2.6.1 Deprecated and Removed Interfaces	19			
		2.6.2 Fortran Binding Issues	20			
		2.6.3 C Binding Issues	21			
		2.6.4 Functions and Macros	21			
	2.7	Processes	22			
	2.8	Error Handling	22			

	2.9	Impler	nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Exam	ples $\ldots \ldots 24$
•			
3			Point Communication 25
	3.1		uction $\ldots \ldots 25$
	3.2		ng Send and Receive Operations
		3.2.1	Blocking Send
		3.2.2	Message Data
		3.2.3	Message Envelope
		3.2.4	Blocking Receive
		3.2.5	Return Status 32
		3.2.6	Passing MPI_STATUS_IGNORE for Status
	3.3		Fype Matching and Data Conversion 35
		3.3.1	Type Matching Rules 35
			Type MPI_CHARACTER 36
		3.3.2	Data Conversion
	3.4		unication Modes
	3.5		tics of Point-to-Point Communication
	3.6	Buffer	Allocation and Usage
		3.6.1	Model Implementation of Buffered Mode. 48
	3.7		ocking Communication
		3.7.1	Communication Request Objects
		3.7.2	Communication Initiation
		3.7.3	Communication Completion
		3.7.4	Semantics of Nonblocking Communications
		3.7.5	Multiple Completions 59
		3.7.6	Non-destructive Test of status
	3.8	Probe	and Cancel
		3.8.1	Probe
		3.8.2	Matching Probe
		3.8.3	Matched Receives
		3.8.4	Cancel
	3.9	Persist	cent Communication Requests
	3.10	Send-I	Receive
	3.11	Null P	Processes
	.		
4		atypes	
	4.1		d Datatypes
		4.1.1	Type Constructors with Explicit Addresses 87
		4.1.2	Datatype Constructors 87 State State
		4.1.3	Subarray Datatype Constructor 96 Distribute 0
		4.1.4	Distributed Array Datatype Constructor
		4.1.5	Address and Size Functions
		4.1.6	Lower-Bound and Upper-Bound Markers 106
		4.1.7	Extent and Bounds of Datatypes
		4.1.8	True Extent of Datatypes 110

		4.1.9	Commit and Free			111
		4.1.10	Duplicating a Datatype			113
		4.1.11	Use of General Datatypes in Communication			113
		4.1.12	Correct Use of Addresses			117
		4.1.13	Decoding a Datatype			118
			Examples			125
	4.2		and Unpack			134
	4.3		ical MPI_PACK and MPI_UNPACK			140
5	Coll	ective	Communication			143
	5.1	Introd	uction and Overview			143
	5.2	Comm	nunicator Argument			146
		5.2.1	Specifics for Intracommunicator Collective Operations			146
		5.2.2	Applying Collective Operations to Intercommunicators			147
		5.2.3	Specifics for Intercommunicator Collective Operations			148
	5.3	Barrie	r Synchronization			149
	5.4		cast			150
		5.4.1	Example using MPI_BCAST			151
	5.5	Gather	\mathbf{r}			151
		5.5.1	Examples using MPI_GATHER, MPI_GATHERV			154
	5.6	Scatter	\mathbf{r}			161
		5.6.1	Examples using MPI_SCATTER, MPI_SCATTERV			164
	5.7		r-to-all			167
		5.7.1	Example using MPI_ALLGATHER			169
	5.8		-All Scatter/Gather			170
	5.9		l Reduction Operations			175
	0.0	5.9.1	Reduce			176
		5.9.2	Predefined Reduction Operations			178
		5.9.3	Signed Characters and Reductions			180
		5.9.4	MINLOC and MAXLOC			181
		5.9.5	User-Defined Reduction Operations			185
		0.0.0	Example of User-defined Reduce			188
		5.9.6	All-Reduce			
		5.9.7	Process-Local Reduction			191
	5 10		e-Scatter			192
	0.10		MPI_REDUCE_SCATTER_BLOCK			192
			MPI_REDUCE_SCATTER			194
	5 11					194
	0.11		Inclusive Scan			195
			Exclusive Scan			196
			Example using MPI_SCAN			197
	5 1 2		ocking Collective Operations			199
	0.14		Nonblocking Barrier Synchronization			201
			Nonblocking Broadcast			201
		0.12.2	Example using MPI_IBCAST			202
		5 19 9				202
			Nonblocking Gather			$203 \\ 205$
			Nonblocking Scatter			
		0.12.0	Nonblocking Gather-to-all	• •	• •	207

		5.12.6	Nonblocking All-to-All Scatter/Gather	209
		5.12.7	Nonblocking Reduce	212
		5.12.8	Nonblocking All-Reduce	213
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks	214
		5.12.10	0 Nonblocking Reduce-Scatter	215
		5.12.11	1 Nonblocking Inclusive Scan	216
		5.12.12	2 Nonblocking Exclusive Scan	217
	5.13	Persist	tent Collective Operations	217
			Persistent Barrier Synchronization	219
			Persistent Broadcast	219
		5.13.3	Persistent Gather	220
		5.13.4	Persistent Scatter	222
		5.13.5	Persistent Gather-to-all	224
		5.13.6	Persistent All-to-All Scatter/Gather	226
			Persistent Reduce	229
			Persistent All-Reduce	230
			Persistent Reduce-Scatter with Equal Blocks	231
			0 Persistent Reduce-Scatter	232
		5.13.11	1 Persistent Inclusive Scan	233
		5.13.12	2 Persistent Exclusive Scan	234
	5.14		tness	234
6	Gro		Contexts, Communicators, and Caching	243
	6.1	Introd	uction	243
		6.1.1	Features Needed to Support Libraries	243
		6.1.2	MPI's Support for Libraries	244
	6.2		Concepts	246
		6.2.1	Groups	246
		6.2.2	Contexts	246
		6.2.3	Intra-Communicators	247
		6.2.4	Predefined Intra-Communicators	247
	6.3	÷	Management	248
		6.3.1	Group Accessors	248
		6.3.2	Group Constructors	250
		6.3.3	Group Destructors	255
	6.4		nunicator Management	255
		6.4.1	Communicator Accessors	255
		6.4.2	Communicator Constructors	257
		6.4.3	Communicator Destructors	268
		6.4.4	Communicator Info	269
	6.5	Motiva	ating Examples	271
		6.5.1	Current Practice $\#1$	271
		6.5.2	Current Practice $#2$	272
		6.5.3	(Approximate) Current Practice #3	272
		6.5.4	Example #4 \ldots	273
		6.5.5	Library Example $\#1$	274
		6.5.6	Library Example $#2$	276
	6.6	Inter-O	Communication	278

		6.6.1	Inter-communicator Accessors
		6.6.2	Inter-communicator Operations
		6.6.3	Inter-Communication Examples
			Example 1: Three-Group "Pipeline"
			Example 2: Three-Group "Ring"
	6.7	Cachin	g
		6.7.1	Functionality
		6.7.2	Communicators
		6.7.3	Windows
		6.7.4	Datatypes
		6.7.5	Error Class for Invalid Keyval
		6.7.6	Attributes Example
	6.8	Namin	g Objects
	6.9	Formal	lizing the Loosely Synchronous Model
		6.9.1	Basic Statements
		6.9.2	Models of Execution
			Static Communicator Allocation
			Dynamic Communicator Allocation
			The General Case
_	-	-	
7			opologies 309
	7.1		309
	7.2		$1 \text{ Topologies } \dots $
	7.3		ding in MPI $\ldots \ldots 310$
	7.4		ew of the Functions $\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 310$
	7.5	-	$gy Constructors \dots \dots$
		7.5.1	Cartesian Constructor
		7.5.2	Cartesian Convenience Function: MPI_DIMS_CREATE
		7.5.3	Graph Constructor
		7.5.4	Distributed Graph Constructor
		7.5.5	Topology Inquiry Functions 322
		7.5.6	Cartesian Shift Coordinates
		7.5.7	Partitioning of Cartesian Structures
		7.5.8	Low-Level Topology Functions
	7.6	-	orhood Collective Communication
		7.6.1	Neighborhood Gather
		7.6.2	Neighbor Alltoall
	7.7		ocking Neighborhood Communication
		7.7.1	Nonblocking Neighborhood Gather
		7.7.2	Nonblocking Neighborhood Alltoall
	7.8		ent Neighborhood Communication
		7.8.1	Persistent Neighborhood Gather
		7.8.2	Persistent Neighborhood Alltoall 351
	7.9	An Ap	plication Example

		Envir	onmental Management	359
	8.1	Impler	nentation Information	359
		8.1.1	Version Inquiries	359
		8.1.2	Environmental Inquiries	360
			Tag Values	361
			Host Rank	361
			IO Rank	361
			Clock Synchronization	362
			Inquire Processor Name	362
	8.2	Memor	ry Allocation	363
	8.3		Handling	366
		8.3.1	Error Handlers for Communicators	368
		8.3.2	Error Handlers for Windows	370
		8.3.3	Error Handlers for Files	371
		8.3.4	Freeing Errorhandlers and Retrieving Error Strings	373
	8.4		Codes and Classes	374
	8.5		Classes, Error Codes, and Error Handlers	
	8.6		s and Synchronization	380
	8.7	Startu	-	381
	0	8.7.1	Allowing User Functions at Process Termination	387
		8.7.2	Determining Whether MPI Has Finished	
	8.8		ble MPI Process Startup	
•				
9	The	Info C	Dbject	391
10	Pro	$\mathbf{cess} \ \mathbf{C}$	reation and Management	397
				001
	10.1	Introd	uction	397
			uction	397
		The D		397
		The D 10.2.1	ynamic Process Model	397 398
	10.2	The D 10.2.1 10.2.2	ynamic Process Model Starting Processes	397 398 398
	10.2	The D 10.2.1 10.2.2 Proces	ynamic Process Model . Starting Processes . The Runtime Environment .	397 398 398 398
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1	ynamic Process Model . Starting Processes . The Runtime Environment . s Manager Interface .	397 398 398 398 398 400
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2	ynamic Process Model	397 398 398 398 400 400
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3	ynamic Process Model . Starting Processes . The Runtime Environment . s Manager Interface . Processes in MPI . Starting Processes and Establishing Communication .	397 398 398 398 400 400 400
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4	ynamic Process Model	397 398 398 398 400 400 400 405
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4	ynamic Process Model	397 398 398 398 400 400 400 400 405 408
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5	ynamic Process Model.Starting Processes.The Runtime Environment.s Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.	397 398 398 398 400 400 400 400 405 408 409
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ	ynamic Process Model.Starting Processes.The Runtime Environment.s Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.	397 398 398 398 400 400 400 400 400 405 408 409 409
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1	ynamic Process Model	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 409\\ 411 \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1 10.4.2	ynamic Process Model.Starting Processes.The Runtime Environment.s Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.Names, Addresses, Ports, and All That.	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 409\\ 411\\ 411\end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ {\rm 10.2.1} \\ {\rm 10.2.2} \\ {\rm Proces} \\ {\rm 10.3.1} \\ {\rm 10.3.2} \\ {\rm 10.3.3} \\ {\rm 10.3.4} \\ {\rm 10.3.5} \\ \\ {\rm Establ} \\ {\rm 10.4.1} \\ {\rm 10.4.2} \\ {\rm 10.4.3} \end{array}$	ynamic Process Model	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 409\\ 411\\ 411\\ 411\\ 412 \end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \end{array}$	ynamic Process Model.Starting Processes.The Runtime Environment.s Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.ishing Communication.Names, Addresses, Ports, and All That.Server Routines.Client Routines.	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 411\\ 411\\ 412\\ 414\end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model.Starting Processes.The Runtime Environment.s Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.ishing Communication.Names, Addresses, Ports, and All That.Server Routines.Name Publishing.Reserved Key Values.	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 409\\ 411\\ 411\\ 412\\ 414\\ 416\\ \end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 409\\ 411\\ 411\\ 411\\ 412\\ 414\\ 416\\ 418\\ \end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model.Starting Processes.The Runtime Environment.s Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.ishing Communication.Names, Addresses, Ports, and All That.Server Routines.Name Publishing.Reserved Key Values.Simplest Example.Simplest Example.	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 411\\ 411\\ 412\\ 414\\ 416\\ 418\\ 418\\ 418\\ \end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model	$\begin{array}{c} 397\\ 398\\ 398\\ 398\\ 398\\ 400\\ 400\\ 400\\ 400\\ 405\\ 408\\ 409\\ 409\\ 411\\ 411\\ 412\\ 414\\ 416\\ 418\\ 418\\ 418\\ 418\\ 418\\ \end{array}$

	10.5.1 Universe Size	421
	10.5.2 Singleton MPI_INIT	422
	10.5.3 MPI_APPNUM	422
	10.5.4 Releasing Connections	423
	10.5.5 Another Way to Establish MPI Communication	425
11 One	-Sided Communications 4	427
11.1	Introduction	427
11.2	Initialization	428
	11.2.1 Window Creation	429
	11.2.2 Window That Allocates Memory	431
	11.2.3 Window That Allocates Shared Memory	433
	11.2.4 Window of Dynamically Attached Memory	436
	11.2.5 Window Destruction	439
	11.2.6 Window Attributes	440
	11.2.7 Window Info	441
11.3	Communication Calls	443
	11.3.1 Put	444
	11.3.2 Get	446
	11.3.3 Examples for Communication Calls	447
	11.3.4 Accumulate Functions	449
	Accumulate Function	450
	Get Accumulate Function	452
	Fetch and Op Function	453
	Compare and Swap Function	455
	11.3.5 Request-based RMA Communication Operations	456
		461
11.5	Synchronization Calls	462
	11.5.1 Fence	466
		467
		471
		474
		476
	11.5.6 Miscellaneous Clarifications	478
11.6		478
		478
		478
11.7		479
		487
		487
	8	488
		490
11.8	Examples	490

12	Exte	ernal I	nterfaces						501
	12.1	Introd	uction		 		 •		501
	12.2	Genera	alized Requests		 				501
		12.2.1	Examples		 				506
	12.3	Associ	ating Information with Status		 				508
	12.4	MPI ai	nd Threads		 				510
			General						
		12.4.2	Clarifications		 				511
		12.4.3	Initialization		 				513
13	I/O								517
	13.1	Introd	uction		 		 •		
			Definitions						517
	13.2	File M	anipulation		 		 •		519
		13.2.1	Opening a File		 		 •		519
		13.2.2	Closing a File		 				521
		13.2.3	Deleting a File		 				522
		13.2.4	Resizing a File		 				523
		13.2.5	Preallocating Space for a File		 				524
		13.2.6	Querying the Size of a File		 				524
		13.2.7	Querying File Parameters		 				525
		13.2.8	File Info		 				526
			Reserved File Hints		 				528
	13.3	File V	iews		 				530
	13.4	Data A	Access		 				533
		13.4.1	Data Access Routines		 				533
			Positioning		 				533
			Synchronism						
			Coordination						
			Data Access Conventions		 				535
		13.4.2	Data Access with Explicit Offsets		 				535
		13.4.3	Data Access with Individual File Pointers .		 				540
		13.4.4	Data Access with Shared File Pointers		 				548
			Noncollective Operations		 				549
			Collective Operations						
			Seek						
		13.4.5							554
	13.5	File In	teroperability						
			Datatypes for File Interoperability						
			External Data Representation: "external32"						
			User-Defined Data Representations						
			Extent Callback						
			Datarep Conversion Functions						
		13.5.4	Matching Data Representations						570
	13.6		tency and Semantics						570
	10.0		File Consistency						
			Random Access vs. Sequential Files						
			Progress						
				-		-		-	

	13.6.4	Collective File Operations	. 574
	13.6.5	Nonblocking Collective File Operations	. 574
		Type Matching	
	13.6.7	Miscellaneous Clarifications	. 575
		MPI_Offset Type	
		Logical vs. Physical File Layout	
		OFile Size	
		1 Examples	
		Asynchronous I/O	
13.7	I/O E	rror Handling	
		rror Classes	
		ples	
		Double Buffering with Split Collective I/O	
		Subarray Filetype Constructor	
14 Too	l Supp	sort	587
		uction	
		ng Interface	
17.4		Requirements	
		Discussion	
		Logic of the Design	
		Miscellaneous Control of Profiling	
		Profiler Implementation Example	
		MPI Library Implementation Example	
	14.2.0	Systems with Weak Symbols	
	1497	Systems Without Weak Symbols	
	14.2.1	Complications	
		Multiple Counting	
		Linker Oddities	
	1400	Fortran Support Methods	
14.0		Multiple Levels of Interception	
14.3		IPI Tool Information Interface	
		Verbosity Levels	
		Binding MPI Tool Information Interface Variables to MPI Objects	
		Convention for Returning Strings	
		Initialization and Finalization	
		Datatype System	
	14.3.6	Control Variables	
		Control Variable Query Functions	
		Example: Printing All Control Variables	
		Handle Allocation and Deallocation	
		Control Variable Access Functions	
		Example: Reading the Value of a Control Variable	
	14.3.7	Performance Variables	
		Performance Variable Classes	
		Performance Variable Query Functions	
		Performance Experiment Sessions	
		Handle Allocation and Deallocation	. 611

	Starting and Stopping of Performance Variables	613
	Performance Variable Access Functions	614
	Example: Tool to Detect Receives with Long Unexpected Message	
	Queues	616
14.3.8	Variable Categorization	618
	Category Query Functions	619
		621
14.3.9		622
		623
15 D		00 5
15 Deprecate		625
	cated since MPI-2.0	625
		628
15.3 Depree	cated since MPI-3.2	628
16 Removed	Interfaces	629
16.1 Remov	red MPI-1 Bindings	629
16.1.1	Overview	629
16.1.2	Removed MPI-1 Functions	629
16.1.3	Removed MPI-1 Datatypes	629
	Removed MPI-1 Constants	629
		630
		630
		0.0.1
	1	631
17.1 Backw	ard Incompatible since MPI-3.2	631
18 Language	Bindings	633
18.1 Fortra	n Support	633
18.1.1	Overview	633
18.1.2	Fortran Support Through the mpi_f08 Module	634
	Fortran Support Through the mpi Module	637
1015	Fortran Support Through the mpif.h Include File	639
10.1.0		$\begin{array}{c} 639\\ 640 \end{array}$
	Interface Specifications, Procedure Names, and the Profiling Interface	640
18.1.6	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	
18.1.6 18.1.7	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	$640 \\ 645 \\ 649$
$18.1.6 \\18.1.7 \\18.1.8$	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	$640 \\ 645 \\ 649 \\ 650$
$18.1.6 \\18.1.7 \\18.1.8$	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	$640 \\ 645 \\ 649 \\ 650 \\ 651$
$18.1.6 \\18.1.7 \\18.1.8$	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652
$18.1.6 \\18.1.7 \\18.1.8$	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652 655
18.1.6 18.1.7 18.1.8 18.1.9	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652 655 658
18.1.6 18.1.7 18.1.8 18.1.9	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652 655 658 659
18.1.6 18.1.7 18.1.8 18.1.9 18.1.10 18.1.11	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652 655 658 659
18.1.6 18.1.7 18.1.8 18.1.9 18.1.10 18.1.11	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	$640 \\ 645 \\ 649 \\ 650 \\ 651 \\ 652 \\ 655 \\ 658 \\ 659 \\ 661$
18.1.6 18.1.7 18.1.8 18.1.9 18.1.10 18.1.11 18.1.12	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652 655 658
18.1.6 18.1.7 18.1.8 18.1.9 18.1.10 18.1.11 18.1.12	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	640 645 649 650 651 652 655 658 659 661 661
18.1.6 18.1.7 18.1.8 18.1.9 18.1.10 18.1.11 18.1.12 18.1.13	Interface Specifications, Procedure Names, and the Profiling Interface MPI for Different Fortran Standard Versions	

	18.1.1	5 Fortran Derived Types	665
	18.1.10	6 Optimization Problems, an Overview	667
	18.1.1'	7 Problems with Code Movement and Register Optimization	668
		Nonblocking Operations	668
		Persistent Operations	669
		One-sided Communication	669
		MPI_BOTTOM and Combining Independent Variables in Datatypes	669
			669
		The Fortran ASYNCHRONOUS Attribute	671
		Calling MPI_F_SYNC_REG	672
		0	673
			674
			674
			674
	18.1.18		674
			676
			676
18			681
10.			681
			681
			681
			682
			684
			686
	16.2.0		687
			688
			688
	1007		689 689
			689
			693
			693
	18.2.10	OInterlanguage Communication	694
Δ Τ ο	nguago	Bindings Summary	697
			697
л.1	A.1.1		697
	A.1.1 A.1.2		710
	A.1.2 A.1.3		710 712
	A.1.3		
		6	712
		Ŭ .	712
	A 1 4		715
	A.1.4		717
	A.1.5		718
	A.1.6		718
A.2			719
A.3		0	722
	A.3.1	0	722
	A.3.2	Datatypes C Bindings	724

		A.3.3 Collective Communication C Bindings	726
		A.3.4 Groups, Contexts, Communicators, and Caching C Bindings	730
		A.3.5 Process Topologies C Bindings	732
		A.3.6 MPI Environmental Management C Bindings	735
		A.3.7 The Info Object C Bindings	736
		A.3.8 Process Creation and Management C Bindings	736
		A.3.9 One-Sided Communications C Bindings	737
		A.3.10 External Interfaces C Bindings	739
		A.3.11 I/O C Bindings	740
		A.3.12 Language Bindings C Bindings	742
		A.3.13 Tools / Profiling Interface C Bindings	743
		A.3.14 Tools / MPI Tool Information Interface C Bindings	743
		A.3.15 Deprecated C Bindings	745
	A.4	Fortran 2008 Bindings with the mpi_f08 Module	746
		A.4.1 Point-to-Point Communication Fortran 2008 Bindings	746
		A.4.2 Datatypes Fortran 2008 Bindings	751
		A.4.3 Collective Communication Fortran 2008 Bindings	756
		A.4.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings	
		A.4.5 Process Topologies Fortran 2008 Bindings	774
		A.4.6 MPI Environmental Management Fortran 2008 Bindings	780
		A.4.7 The Info Object Fortran 2008 Bindings	782
		A.4.8 Process Creation and Management Fortran 2008 Bindings	783
		A.4.9 One-Sided Communications Fortran 2008 Bindings	785
		A.4.10 External Interfaces Fortran 2008 Bindings	790
		A.4.11 I/O Fortran 2008 Bindings	791
		A.4.12 Language Bindings Fortran 2008 Bindings	799
		A.4.13 Tools / Profiling Interface Fortran 2008 Bindings	800
	A.5	Fortran Bindings with mpif.h or the mpi Module	801
		A.5.1 Point-to-Point Communication Fortran Bindings	801
		A.5.2 Datatypes Fortran Bindings	804
		A.5.3 Collective Communication Fortran Bindings	806
		A.5.4 Groups, Contexts, Communicators, and Caching Fortran Bindings	812
		A.5.5 Process Topologies Fortran Bindings	816
		A.5.6 MPI Environmental Management Fortran Bindings	819
		A.5.7 The Info Object Fortran Bindings	822
		A.5.8 Process Creation and Management Fortran Bindings	822
		A.5.9 One-Sided Communications Fortran Bindings	823
		A.5.10 External Interfaces Fortran Bindings	827
		A.5.11 I/O Fortran Bindings	828
		A.5.12 Language Bindings Fortran Bindings	833
		A.5.13 Tools / Profiling Interface Fortran Bindings	833
		A.5.14 Deprecated Fortran Bindings	833
			200
В	Cha	nge-Log	835
	B.1	Changes from Version 3.1 to Version 3.2	835
		B.1.1 Changes in MPI-3.2	835
	B.2	Changes from Version 3.0 to Version 3.1	836
		B.2.1 Fixes to Errata in Previous Versions of MPI	836

	B.2.2 Changes in MPI-3.1	838			
B.3	Changes from Version 2.2 to Version 3.0	839			
	B.3.1 Fixes to Errata in Previous Versions of MPI	839			
	B.3.2 Changes in MPI-3.0	840			
B.4	Changes from Version 2.1 to Version 2.2	845			
B.5	Changes from Version 2.0 to Version 2.1	847			
Bibliog	graphy	853			
Genera	al Index	858			
Examples Index					
MPI C	Constant and Predefined Handle Index	866			
MPI Declarations Index					
MPI Callback Function Prototype Index					
MPI Function Index					

List of Figures

5.1	Collective communications, an overview	145
5.2	Intercommunicator allgather	148
5.3	Intercommunicator reduce-scatter	149
5.4	Gather example	155
5.5	Gatherv example with strides	156
5.6	Gatherv example, 2-dimensional	157
5.7	Gatherv example, 2-dimensional, subarrays with different sizes	158
5.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	160
5.9	Scatter example	165
5.10		165
5.11	Scattery example with different strides and counts	166
5.12	Race conditions with point-to-point and collective communications	237
5.13	Overlapping Communicators Example	241
6.1	0 – –	262
6.2		266
6.3	· · · · ·	284
6.4	Three-group ring	285
7.1	Neighborhood gather communication example.	336
7.2	0 0 i	355
7.3	Communication routine with local data copying and sparse neighborhood	000
1.0		356
7.4	Communication routine with sparse neighborhood all-to-all-w and without	000
		357
7.5	Two-dimensional parallel Poisson solver with persistent sparse neighborhood	
		358
11.1	Schematic description of the public/private window operations in the	
		462
	8	464
11.3	Active target communication, with weak synchronization	465
11.4	Passive target communication	466
11.5	Active target communication with several processes	470
		488
11.7	Deadlock situation	489
		489
10.1		F 1 0
13.1	Etypes and filetypes	518

13.2	Partitioning a file among parallel processes															518
13.3	Displacements															531
13.4	Example array file layout															584
13.5	Example local array file type for process 1 $% \left(1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2$	•	•	•	•	•	 •	•	•	•	•		•	•	•	585
18.1	Status conversion routines										•		•			685

List of Tables

11	2.1	Deprecated and Removed constructs	20
12 13	3.1	Predefined MPI datatypes corresponding to Fortran datatypes	27
14	3.2	Predefined MPI datatypes corresponding to C datatypes	28
15	3.3	Predefined MPI datatypes corresponding to both C and Fortran datatypes .	29
16	3.4	Predefined MPI datatypes corresponding to C++ datatypes	29
17 18	4.1	combiner values returned from MPI_TYPE_GET_ENVELOPE $\ldots \ldots \ldots$	119
19 20	6.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	280
21	8.1	Error classes (Part 1)	375
22	8.2	Error classes (Part 2)	376
23	11 1	C types of attribute value argument to MPI_WIN_GET_ATTR and	
24	11.1	MPI_WIN_SET_ATTR.	440
25	11.9	Error classes in one-sided communication routines	478
26	11.4		110
27	13.1	Data access routines	533
28	13.2	"external32" sizes of predefined datatypes	566
29	13.3	I/O Error Classes	582
30			
31		MPI tool information interface verbosity levels	594
32		Constants to identify associations of variables	595
33		MPI datatypes that can be used by the MPI tool information interface	597
34		Scopes for control variables	601
35	14.5	Return codes used in functions of the MPI tool information interface	624
36	16.1	Removed MPI-1 functions and their replacements	629
37		Removed MPI-1 datatypes and their replacements	630
38		Removed MPI-1 constants	630
39		Removed MPI-1 callback prototypes and their replacements	630
40	10.4	tomoved in Fr candaex prototypes and then replacements	000
41	18.1	Specific Fortran procedure names and related calling conventions	641
42		Occurrence of Fortran optimization problems	667
43		* *	
44			
45			

Acknowledgments

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

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

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

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

18 19• Jack Dongarra, David Walker, Conveners and Meeting Chairs 2021• Ewing Lusk, Bob Knighten, Minutes 22 • Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication 23 24 • Al Geist, Marc Snir, Steve Otto, Collective Communication 2526• Steve Otto, Editor 27• Rolf Hempel, Process Topologies 2829 • Ewing Lusk, Language Binding 30 31• William Gropp, Environmental Management 32 • James Cownie, Profiling 33 34 • Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, 35Contexts, and Communicators 36 37 • Steven Huss-Lederman, Initial Implementation Subset 38 39

The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

40

6

 $\overline{7}$

8

9 10 11

12

13

14

15

16

1	Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz				
2	Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster				
3	Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower				
4	Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison				
5	Leslie Hart	Tom Haupt	Don Heller	Tom Henderson				
6	Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga				
7	James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe				
8	Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley				
9	Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer				
10	Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison				
11	Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson				
12	Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith				
13								
14	0		0	atory made the draft avail-				
15				distributing the document.				
16		-		RPA and NSF under grant				
17				nology Center Cooperative				
18			mmission of the Eur	opean Community through				
19	Esprit project P6643 (Pl	PPE).						
20								
21 22	MPI-1.2 and MPI-2.0:							
23	Those who served as prin	mary coordinators in	n MPI-1.2 and MPI-	2.0 are:				
24								
25	• Ewing Lusk, Conve	ener and Meeting Cl	hair					
26 27	• Steve Huss-Lederm	an, Editor						
28 29	• Ewing Lusk, Misce	llany						
30	• Bill Saphir, Proces	s Creation and Man	agement					
31 32	• Marc Snir, One-Sided Communications							
33 34	• Bill Gropp and An	thony Skjellum, Ext	cended Collective O	perations				
35	• Steve Huss-Lederm	aan, External Interfa	aces					
36 37	• Bill Nitzberg, I/O							
38 39	• Andrew Lumsdaine	, <u> </u>	x v y v	ge Bindings				
40 41	• Anthony Skjellum	and Arkady Kanevs	ky, Real-Time					
42 43	The following list in meetings and are not me		active participants w	vho attended MPI-2 Forum				
43	-							
45								
46								
47								
48								
-								

	Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar	1
	Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz	2
	Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen	3
	Ying Chen	Albert Cheng	Yong Cho	Joel Clark	4
	Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie	5
	Zhenqian Cui	Suresh Damodaran-Kar	nal	Raja Daoud	6
	Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra	7
	Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon	8
	Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman	9
	Ian Foster	Hubertus Franke	Richard Frost	Al Geist	10
	Robert George	David Greenberg	John Hagedorn	Kei Harada	11
	Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson	12
	Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones	13
	Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica	14
	Steve Landherr	Mario Lauria	Mark Law	Juan Leon	15
	Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams	16
	John May	Oliver McBryan	Brian McCandless	Tyce McLarty	17
	Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha	18
	Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco	19
	Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce	20
	Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov	21
	James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee	22
	Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha	23
	Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman	24
	Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale	25
	David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur	26
	Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon	27
	David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong	28
	Dave Wright			Ū	29
	U				30
		also acknowledges and ap	preciates the valuable	e input from people via	31
e	-mail and in person.				32
		titutions supported the N	/IPI-2 effort through t	time and travel support	33
f	or the people listed a	bove.			34
	Argonne National	Laboratory			35
	Bolt, Beranek, an	-			36
	California Institu				37
	Center for Compu	90			38
	Convex Compute				39
	Cray Research	I I I I I I I I I I I I I I I I I I I			40
	Digital Equipmen	t Corporation			41
		nect Solutions, Inc.			42
	_	el Computing Centre			43
	General Electric (44
		Descende Conton for Infe	mostion Technology		45

Hewlett-Packard Hitachi

Hughes Aircraft Company

German National Research Center for Information Technology

Unofficial Draft for Comment Only

xxiii

 $45 \\ 46$

47

1	Intel Corporation
2	International Business Machines
3	Khoral Research
4	Lawrence Livermore National Laboratory
5	Los Alamos National Laboratory
6	MPI Software Techology, Inc.
7	Mississippi State University
8	NEC Corporation
9	National Aeronautics and Space Administration
10	National Energy Research Scientific Computing Center
11	National Institute of Standards and Technology
12	National Oceanic and Atmospheric Administration
13	Oak Ridge National Laboratory
14	The Ohio State University
15	PALLAS GmbH
16	Pacific Northwest National Laboratory
17	Pratt & Whitney
18	San Diego Supercomputer Center
19	Sanders, A Lockheed-Martin Company
20	Sandia National Laboratories
21	Schlumberger
22	Scientific Computing Associates, Inc.
23	Silicon Graphics Incorporated
24	Sky Computers
25	Sun Microsystems Computer Corporation
26	Syracuse University
27	The MITRE Corporation
28	Thinking Machines Corporation
29	United States Navy
30	University of Colorado
31	University of Denver
32	University of Houston
33	University of Illinois
34	University of Maryland
35	University of Notre Dame
36	University of San Fransisco
37	University of Stuttgart Computing Center
38	University of Wisconsin
39	·
40	MPI-2 operated on a very tight budget (in reality, it had no budget when the first
41	meeting was announced). Many institutions helped the MPI-2 effort by supporting the
42	efforts and travel of the members of the MPI Forum. Direct support was given by NSF and
43	DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and
44	Esprit under project HPC Standards (21111) for European participants.
45	
46	
47	
48	

MPI-1.3 and MPI-2.1:	1
The editors and organizers of the combined documents have been:	2 3
• Richard Graham, Convener and Meeting Chair	4
• Jack Dongarra, Steering Committee	5 6
• Al Geist, Steering Committee	7
• Bill Gropp, Steering Committee	8 9
Rainer Keller, Merge of MPI-1.3	10
, 0	11 12
• Andrew Lumsdaine, Steering Committee	13
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)	14 15 16
• Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)	17 18
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:	19 20 21
• Bill Gropp, Front matter, Introduction, and Bibliography	22
• Richard Graham, Point-to-Point Communication	23 24
• Adam Moody, Collective Communication	25 26
• Richard Treumann, Groups, Contexts, and Communicators	27
• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions	28 29 30
• George Bosilca, Environmental Management	31 32
• David Solt, Process Creation and Management	33
• Bronis R. de Supinski, External Interfaces, and Profiling	34 35
• Rajeev Thakur, I/O	36
 Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary 	37 38
	39
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log	40 41
• Alexander Supalov and Denis Nagorny, Annex Language Bindings	42
The following list includes some of the active participants who attended MPI-2 Forum	43
meetings and in the e-mail discussions of the errata items and are not mentioned above.	44 45
	46
	47
	48

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Jeffrey Brown
4	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
5	Terry Dontje	Gabor Dozsa	Edric Ellis
6	Karl Feind	Edgar Gabriel	Patrick Geoffray
7	David Gingold	Dave Goodell	Erez Haba
8	Robert Harrison	Thomas Herault	Steve Hodson
9	Torsten Hoefler	Joshua Hursey	Yann Kalemkarian
10	Matthew Koop	Quincey Koziol	Sameer Kumar
11	Miron Livny	Kannan Narasimhan	Mark Pagel
12	Avneesh Pant	Steve Poole	Howard Pritchard
13	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
14	Tony Skjellum	Brian Smith	Vinod Tipparaju
15	Jesper Larsson Träff	Keith Underwood	
16			
17		nowledges and appreciates the	valuable input from people via
18	e-mail and in person.		
19	=	s supported the MPI-2 effort t	hrough time and travel support
20	for the people listed above.		
21	Argonne National Labora	atory	
22	Bull		
23	Cisco Systems, Inc.		
24	Cray Inc.		
25	The HDF Group		
26	Hewlett-Packard		
27	IBM T.J. Watson Resear	ch	
28	Indiana University		
29		nerche en Informatique et Aut	omatique (INRIA)
30	Intel Corporation		
31	Lawrence Berkeley Natio	-	
32 33	Lawrence Livermore Nat	°	
34	Los Alamos National Lab	poratory	
35	Mathworks		
36	Mellanox Technologies Microsoft		
37	Myricom		
38	NEC Laboratories Europ	o NEC Europo I td	
39	Oak Ridge National Lab	· -	
40	The Ohio State Universit	0	
41	Pacific Northwest Nation		
42	QLogic Corporation	ar Laboratory	
43	Sandia National Laborat	ories	
44	SiCortex	01165	
45	Silicon Graphics Incorpor	rated	
46	Sun Microsystems, Inc.		
47	University of Alabama at	Birmingham	
48	University of Houston		
	Chiverency of Houston		

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

_			
1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
2	Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Greg Bronevetsky
4	Jeff Brown	Darius Buntinas	Jonathan Carter
5	Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
6	Edric Ellis	Karl Feind	Edgar Gabriel
7	Patrick Geoffray	Johann George	David Gingold
8	David Goodell	Erez Haba	Robert Harrison
9	Thomas Herault	Marc-André Hermanns	Steve Hodson
10	Joshua Hursey	Yutaka Ishikawa	Bin Jia
11	Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
12	Ranier Keller	Matthew Koop	Quincey Koziol
13	Manojkumar Krishnan	Sameer Kumar	Miron Livny
14	Andrew Lumsdaine	Miao Luo	Ewing Lusk
15	Timothy I. Mattox	Kannan Narasimhan	Mark Pagel
16	Avneesh Pant	Steve Poole	Howard Pritchard
17	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
18	Martin Schulz	Pavel Shamis	Galen Shipman
19	Christian Siebert	Anthony Skjellum	Brian Smith
20	Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
21	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu
22	The MDI Ferrura else estruct	ladges and approxistes the m	hughle input from people vie
23		vledges and appreciates the va	nuable input from people via
24	e-mail and in person.	provide the MPL 2.2 offerst the	augh time and travel support
25	for the people listed above.	pported the MPI-2.2 effort thr	ough time and travel support
26	for the people listed above.		
27	Argonne National Laborato	ry	
28	Auburn University		
29	Bull		
30	Cisco Systems, Inc.		
31	Cray Inc.		
32	Forschungszentrum Jülich		
33	Fujitsu		
34	The HDF Group		
35	Hewlett-Packard		
36	International Business Mach	nines	
37	Indiana University		
38	Institut National de Rechere	che en Informatique et Autom	atique (INRIA)
39	Institute for Advanced Scien	nce & Engineering Corporation	n
40	Intel Corporation		
41	Lawrence Berkeley National	Laboratory	
42	Lawrence Livermore Nation	al Laboratory	
43	Los Alamos National Labor	atory	
44	Mathworks		
45	Mellanox Technologies		
46	Microsoft		
47	Myricom		
48	NEC Corporation		

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

Unofficial Draft for Comment Only

The following list includes some of the active participants who attended MPI-3 Forum 2 meetings or participated in the e-mail discussions and who are not mentioned above.

	mootings of participated in the e in		are not mentioned above.
3 4	Tatsuya Abe	Tomoya Adachi	Sadaf Alam
4 5	Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore
6	Brian Barrett	Richard Barrett	Robert Blackmore
7	Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky
8	Jed Brown	Darius Buntinas	Devendar Bureddy
9	Arno Candel	George Carr	Mohamad Chaarawi
10	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
15	Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa
16	Nysal Jan	Bin Jia	Hideyuki Jitsumoto
17	Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima
18	Chulho Kim	Dries Kimpe	Christof Klausecker
19	Alice Koniges	Quincey Koziol	Dieter Kranzlmueller
20	Manojkumar Krishnan	Sameer Kumar	Eric Lantz
21	Jay Lofstead	Bill Long	Andrew Lumsdaine
22	Miao Luo	Ewing Lusk	Adam Moody
23	Nick M. Maclaren	Amith Mamidala	Guillaume Mercier
24	Scott McMillan	Douglas Miller	Kathryn Mohror
25	Tim Murray	Tomotake Nakamura	Takeshi Nanri
26	Steve Oyanagi	Mark Pagel	Swann Perarnau
27	Sreeram Potluri	Howard Pritchard	Rolf Riesen
28	Hubert Ritzdorf	Kuninobu Sasaki	Timo Schneider
29	Martin Schulz	Gilad Shainer	Christian Siebert
30	Anthony Skjellum	Brian Smith	Marc Snir
31	Raffaele Giuseppe Solca	Shinji Sumimoto	Alexander Supalov
32	Sayantan Sur	Masamichi Takagi	Fabian Tillier
33	Vinod Tipparaju	Jesper Larsson Träff	Richard Treumann
34	Keith Underwood	Rolf Vandevaart	Anh Vo
35	Abhinav Vishnu	Min Xie	Enqiang Zhou
36		1 1	

The MPI Forum also acknowledges and appreciates the valuable input from people via 37e-mail and in person. 38

The MPI Forum also thanks those that provided feedback during the public comment 39 period. In particular, the Forum would like to thank Jeremiah Wilcock for providing detailed 40comments on the entire draft standard. 41

The following institutions supported the MPI-3 effort through time and travel support 42for the people listed above. 43

- Argonne National Laboratory
- 45Bull

44

- 46Cisco Systems, Inc.
- 47Cray Inc.
- 48 CSCS

ETH Zurich	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
	11
Intel Corporation	12
Lawrence Berkeley National Laboratory	13
Lawrence Livermore National Laboratory	14
Los Alamos National Laboratory	14
Mellanox Technologies, Inc.	
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
Funding for the MDI Ferring meetings was partially supported by swands // CCF 0816000	36
Funding for the MPI Forum meetings was partially supported by awards $\#CCF-0816909$	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
MPI-3.1:	41
MPI-3.1 is a minor update to the MPI Standard.	42
The editors and organizers of the MPI-3.1 have been:	43
The editors and organizers of the twi 10.1 have been.	44
• Martin Schulz, MPI-3.1 chair	45
	46
• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu-	47
nications, and Bibliography; Overall editor	48

Unofficial Draft for Comment Only

1 2	• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindi Deprecated Functions, Annex Change-Log, and Annex Language Bindings					
3 4	• Richard L. Graham, Steering committee, Meeting Convener					
5 6	• Jeffrey M. Squyres, Language Bindings and MPI-3.1 Secretary					
7	• Daniel Holmes, Point-to-Point Communication					
8 9	• George Bosilca, Datatypes and Environmental Management					
10 11	• Torsten Hoefler, Collective Communication and Process Topologies					
12	• Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces					
13 14	• Jeff Hammond, The Info Object					
15 16	• David Solt, Process Creation and Management					
17	• Quincey Koziol, I/O					
18 19	• Kathryn Mohror, Tool S	upport				
20 21	• Rajeev Thakur, One-Side	ed Communications				
22 23 24	The following list includer meetings or participated in the	-	rticipants who attended MPI Forum			
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	Brian Barrett George Bosilca Yohann Burette James Dinan Edgar Gabriel Paddy Gillies Richard L. Graham Khaled Hamidouche Marc-André Hermanns Daniel Holmes Hideyuki Jitsumoto Christos Kavouklis Michael Knobloch Sameer Kumar Huiwei Lu Adam Moody Steve Oyanagi Howard Pritchard Ken Raffenetti Davide Rossetti	Wesley Bland Aurelien Bouteiller Mohamad Chaarawi Dmitry Durnov Todd Gamblin David Goodell Ryan E. Grant Jeff Hammond Nathan Hjelm Atsushi Hori Jithin Jose Takahiro Kawashima Alice Koniges Joshua Ladd Guillaume Mercier Tomotake Nakamura Antonio J. Pẽna Rolf Rabenseifner Raghunath Raja Kento Sato Christian Sichart	Michael Blocksome Devendar Bureddy Alexey Cheptsov Thomas Francois Balazs Gerofi Manjunath Gorentla Venkata William Gropp Amin Hassani Torsten Hoefler Yutaka Ishikawa Krishna Kandalla Chulho Kim Quincey Koziol Ignacio Laguna Kathryn Mohror Takeshi Nanri Sreeram Potluri Nicholas Radcliffe Craig Rasmussen Martin Schulz			
46 47	Sangmin Seo Brian Smith	Christian Siebert David Solt	Anthony Skjellum Jeffrey M. Squyres			
48						

	Hari Subramoni	Shinji Sumimoto	Alexander Supalov	1
	Bronis R. de Supinski	Sayantan Sur	Masamichi Takagi	2
	Keita Teranishi	Rajeev Thakur	Fabian Tillier	3
	Yuichi Tsujita	Geoffroy Vallée	Rolf vandeVaart	4
	Akshay Venkatesh	Jerome Vienne	Venkat Vishwanath	5
	Anh Vo	Huseyin S. Yildiz	Junchao Zhang	6
	Xin Zhao			7
				8
		ges and appreciates	the valuable input from people via	9
e-mail and	•			10
		rted the MPI-3.1 effc	rt through time and travel support	11
for the peop	ple listed above.			12
Argonn	ne National Laboratory			13
Aubur	n University			14
Cisco S	Systems, Inc.			15
Cray				16
•	The University of Edinb	ourgh		17
ETH Z		0		18
Forsch	ungszentrum Jülich			19
Fujitsu	0			20
-	n Research School for Sir	nulation Sciences		21
The H	DF Group			22
	ational Business Machines	3		23
INRIA				24
	orporation			25
		e. High-Performance	Computing (JARA-HPC)	26
	u University	, <u>o</u>		27
-	nce Berkeley National La	boratory		28
	nce Livermore National L	-		29
Lenovo		J		30
Los Al	amos National Laborator	V		31
	ox Technologies, Inc.	0		32
	oft Corporation			33
	Corporation			34
	A Corporation			35
	idge National Laboratory			36
	hio State University			37
	N AICS			38
	National Laboratories			39
	Advanced Computing Ce	nter		40
	Institute of Technology			41
v	sity of Alabama at Birmi	nơham		42
	sity of Houston			43
	sity of Illinois at Urbana-	Champaign		44
	sity of Oregon	Champar81		44
		erformance Computi	ng Center Stuttgart (HLRS)	40
	sity of Tennessee, Knoxvi	—	ng center Stutugart (IILIUS)	40
	sity of Tokyo			48
Univer	SIUY OF TORYO			40

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-processors, 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 [50], Express [13], nCUBE's Vertex [46], p4 [8, 9], and PARMACS [5, 10]. Other important contributions have come from Zipcode [53, 54], Chimp [19, 20], PVM [4, 17], Chameleon [27], and PICL [25].

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

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

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

- 44
- $45 \\ 46$

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

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

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.

Background of MPI-1.3 and MPI-2.1 1.4

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

Unofficial Draft for Comment Only

1 2

3

4

56

7

8 9

10 11

12

13

1415

16

17

18

19

20

21

22

23

 24 25

26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

41

43

45

46

1 Restarting regular work of the MPI Forum was initiated in three meetings, at Eu- $\mathbf{2}$ roPVM/MPI'06 in Bonn, at EuroPVM/MPI'07 in Paris, and at SC'07 in Reno. In De-3 cember 2007, a steering committee started the organization of new MPI Forum meetings at 4 regular 8-weeks intervals. At the January 14–16, 2008 meeting in Chicago, the MPI Forum 5decided to combine the existing and future MPI documents to one document for each ver-6 sion of the MPI standard. For technical and historical reasons, this series was started with $\overline{7}$ MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 8 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for 9 MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft document, 10 for each chapter, a chapter author and review team were defined. They cleaned up the 11document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document 12was finished in June 2008, and finally released with a second vote in September 2008 in 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 Background of MPI-3.1

⁴² MPI-3.1 is a minor update to the MPI standard. Most of the updates are corrections ⁴³ and clarifications to the standard, especially for the Fortran bindings. New functions added ⁴⁴ include routines to manipulate MPI_Aint values in a portable manner, nonblocking collective ⁴⁵ I/O routines, and routines to get the index value by name for MPI_T performance and ⁴⁶ control variables. A general index was also added.

47

1.8 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.

1.9 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.

1.10 What Is Included in the Standard?

The standard includes:

• Point-to-point communication,	34
• I onit-to-point communication,	35
• Datatypes,	36
	37
• Collective operations,	38
• Process groups,	39
• Trocess groups,	40
• Communication contexts,	41
	42
• Process topologies,	43
• Environmental management and inquiry,	44
• Environmental management and inquiry,	45
• The Info object,	46
	47
• Process creation and management,	48

Unofficial Draft for Comment Only

 24

• One-sided communication, • External interfaces. • Parallel file I/O, • Language bindings for Fortran and C, • Tool support. What Is Not Included in the Standard? 1.11 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.

Organization of This Document 1.12

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

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point-to-Point Communication, defines the basic, pairwise communication subset of MPI. Send and receive are found here, along with many associated functions designed to make basic communication powerful and efficient.
- Chapter 4, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.
- Chapter 5, Collective Communication, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations.
- Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of pro-46 47 cesses are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.

Unofficial Draft for Comment Only

6

4 56

7

8 9 10

11 12

13

14

1516

1718

19 20

21

22

23

 24

2526

2728

29

30 31

32

33 34

35 36

37

38

39 40

41

42

43

44

45

48

1

 $\mathbf{2}$

- 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 Interfaces, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 16, Removed Interfaces, describes routines and constructs that have been removed from MPI. These were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard.
- Chapter 17, Backward Incompatibilities, describes incompatibilities with previous versions of MPI.
- Chapter 18, 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 46 standard. 47

Unofficial Draft for Comment Only

 24

• 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 bind-ing of these interfaces that will enable interoperability is outside the scope of this document. A separate document consists of ideas that were discussed in the MPI Forum 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

- 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.
 - Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
 - Chapter 4, Communicator ID, describes an approach to providing identifiers for communicators.
 - 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.
 - Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.
 - Chapter 7, Split Collective Communication, describes a specification for certain nonblocking collective operations.
 - Chapter 8, Real-Time MPI, discusses MPI support for real time processing.

 $\mathbf{2}$

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.

2.1 Document Notation

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

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

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

2.2 Naming Conventions

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

- 1. In C, all routines associated with a particular type of MPI object should be of the form MPI_Class_action_subset or, if no subset exists, of the form MPI_Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI_CLASS_ACTION_SUBSET or, if no subset exists, of the form MPI_CLASS_ACTION.
- 2. If the routine is not associated with a class, the name should be of the form MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran.

 $45 \\ 46$

3. The names of certain actions have been standardized. In particular, **Create** creates a new object, **Get** retrieves information about an object, **set** sets this information, **Delete** deletes information, **Is** asks whether or not an object has a certain property.

C and Fortran names for some MPI functions (that were defined during the MPI-1 process) violate these rules in several cases. The most common exceptions are the omission of the **Class** name from the routine and the omission of the **Action** where one can be inferred.

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

2.3 Procedure Specification

MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT, or INOUT. The meanings of these are:

• IN: the call may use the input value but does not update the argument from the perspective of the caller at any time during the call's execution,

• OUT: the call may update the argument but does not use its input value,

• INOUT: the call may both use and update the argument.

There is one special case — if an argument is a handle to an opaque object (these terms are defined in Section 2.5.1), and the object is updated by the procedure call, then the argument is marked INOUT or OUT. It is marked this way even though the handle itself is not modified — we use the INOUT or OUT attribute to denote that what the handle *references* is updated.

28 29

30

31 32 Rationale. The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (End of rationale.)

MPI's use of IN, OUT, and INOUT is intended to indicate to the user how an argument is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., INTENT in Fortran 90 bindings or const in C bindings). For instance, the "constant" MPI_BOTTOM can usually be passed to OUT buffer arguments. Similarly, MPI_STATUS_IGNORE can be passed as the OUT status argument.

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such an argument is, syntactically, an INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

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

⁴⁵ Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased ⁴⁶ with any other argument passed to an MPI procedure. An example of argument aliasing in ⁴⁷ C appears below. If we define a C procedure like this,

48

1

2

3

4

5

6

7 8

9

10

11 12

13 14

15

16

17

18 19

20 21

```
1
void copyIntBuffer( int *pin, int *pout, int len )
                                                                                               2
{
    int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;</pre>
}
then a call to it in the following code fragment has aliased arguments.
                                                                                               6
int a[10];
copyIntBuffer( a, a+3, 7);
                                                                                               9
Although the C language allows this, such usage of MPI procedures is forbidden unless
                                                                                               10
otherwise specified. Note that Fortran prohibits aliasing of arguments.
                                                                                               11
    All MPI functions are first specified in the language-independent notation. Immediately
                                                                                               12
below this, language dependent bindings follow:
                                                                                               13
   • The ISO C version of the function.
                                                                                               14
                                                                                               15
   • The Fortran version used with USE mpi_f08.
                                                                                               16
                                                                                               17
   • The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.
                                                                                               18
    An exception is Section 14.3 "The MPI Tool Information Interface", which only provides
                                                                                               19
ISO C interfaces.
                                                                                               20
    "Fortran" in this document refers to Fortran 90 and higher; see Section 2.6.
                                                                                               21
    The words function, routine, procedure, procedure call, and call are used as synonyms
                                                                                               22
within this standard.
                                                                                               23
                                                                                               ^{24}
                                                                                               25
2.4
      Semantic Terms
                                                                                               26
When discussing MPI procedures the following semantic terms are used.
                                                                                               27
An MPI operation is a set of one or more MPI procedures leading from a well-defined
                                                                                               28
input state to a well-defined output state. An operation consists of four stages: initialization,
                                                                                               29
starting, completion, and freeing:
                                                                                               30
                                                                                               31
Initialization hands over the argument list to the operation but not the content of the
                                                                                               32
     message data buffers. The specification of an operation may state that array argu-
                                                                                               33
     ments must not be changed until the operation is freed.
                                                                                               34
                                                                                               35
Starting hands over the control of the message data buffer to the associated operation.
                                                                                               36
     Note that initiation refers to the combination of the initialization and starting stages.
                                                                                               37
                                                                                               38
Completion returns control of the content of the message data buffer to the application
                                                                                               39
     and indicates that any output buffers have been updated.
                                                                                               40
Freeing returns control of the rest of the argument list (e.g., the buffer address and array
                                                                                               41
     arguments).
                                                                                               42
                                                                                               43
MPI procedures can be complete or incomplete:
                                                                                               44
complete procedure An MPI procedure is complete if return from the procedure
                                                                                               45
                                                                                               46
     indicates that the operation it is associated with is finished, which implies that the
                                                                                               47
     user is allowed to reuse parameters specified in the call. Procedures not associated
                                                                                               48
     with an operation are also defined to be complete.
```

Unofficial Draft for Comment Only

	12 CHAPTER 2. MPI TERMS AND CONVENTIONS
1 2 3 4	incomplete procedure An MPI procedure is incomplete if it returns before the op- eration that it is associated with is finished, which may imply that the user is not allowed to reuse parameters (such as buffers) specified in the call.
5 6 7	In addition to the concept of complete and incomplete, procedures have an orthogonal concept of locality:
8 9 10	local procedure An MPI procedure is local if it returns control to the calling MPI process based only on the state of the local MPI process that invoked it. Local procedures are also characterized as immediate .
11 12 13 14	non-local procedure An MPI procedure is non-local if returning may require the execution by some MPI procedure on another MPI process. Such a procedure may require communication occurring with another MPI process.
$15 \\ 16$	MPI procedures can be blocking or nonblocking:
17 18 19	blocking procedure An MPI procedure is blocking if it is complete or non-local (or both).
20 21	nonblocking procedure An MPI procedure is nonblocking if it is incomplete and local.
22 23 24 25 26 27 28 29	Advice to users. Note that for communication-related procedures, in most cases incomplete procedures are local and complete procedures are non-local. Exceptions are noted where such procedures are defined. In many cases, in the procedure name, an additional prefix letter I as an abbreviation of the words incomplete and immediate marks nonblocking procedures. Some examples for in/complete and non/local procedures are listed here. Nonblocking procedures:
30 31 32 33	 incomplete and local: MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_PUT, MPI_GET, MPI_ACCUMULATE, MPI_IMPROBE, MPI_SEND_INIT, MPI_RECV_INIT,
$34 \\ 35$	Blocking procedures:
36	• complete and nonlocal: MPI_SEND, MPI_RECV, MPI_BCAST, MPI_PROBE,
37 38	 incomplete and nonlocal: MPI_MPROBE, MPI_BCAST_INIT,, MPI_FILE_{READ WRITE}_{AT_ALL ALL ORDERED}_BEGIN.
39 40	• complete and local: MPI_BSEND, MPI_RSEND, MPI_IPROBE, MPI_MRECV.
40 41 42	(End of advice to users.)
43	MPI operations can be blocking, nonblocking, or persistent:
44 45 46 47 48	blocking operation For a blocking operation , all four stages are combined in a single procedure call.

nonblocking operation For a nonblocking operation , the initialization and starting stages are combined into a single nonblocking procedure call and the completion and freeing stages are combined into a separate, single procedure call, which can be block-	1 2 3
ing or nonblocking.	4
persistent operation For a persistent operation , all four stages are done with separate procedure calls, each of which may be blocking or nonblocking.	5 6
procedure cans, each of which may be blocking of honorocking.	7
Additionally, an MPI operation is complete when the completion stage returns.	8 9
Furthermore, a procedure can be collective:	10 11
collective procedure An MPI procedure is collective if all processes in a process group need to invoke the procedure.	12 13 14
collective operations Collective MPI operations are also available as blocking, non- blocking and persistent operations as defined above.	15 16 17
Initialization procedures of collective operations over the same process group must be executed in the same order by all members of the process group.	18 19
The procedures for blocking collective operations and the initialization procedures for persistent collective operations may or may not be synchronizing . That is, they may or may not return before all processes in the group have called the procedure.	20 21 22
The initiation procedures for nonblocking collective operations and the starting proce- dures for persistent collective operations are local and shall not be synchronizing.	23 24 25
In case of nonblocking or persistent collective operations, the completion stage may or may not finish before all processes in the group have started the operation.	26 26 27
Advice to users. Calling any synchronizing function when there is no possibility of corresponding calls at all other processes in the associated process group is erroneous because it can cause deadlock.	28 29 30
Waiting for completion of any operation when there is no possibility that all other processes in the associated group will be able to start the corresponding operation is erroneous because it can cause deadlock. (<i>End of advice to users.</i>)	31 32 33 34
When the words operation and procedure are used, usually MPI operation and MPI procedure are meant.	35 36
Annex A.2 summarizes the semantics of all communicating MPI routines.	37 38
For datatypes, the following terms are defined:	39 40
predefined A predefined datatype is a datatype with a predefined (constant) name (such	41
as MPI_INT, MPI_FLOAT_INT, or MPI_PACKED) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or	42 43
MPI_TYPE_CREATE_F90_COMPLEX. The former are named whereas the latter are	44
unnamed.	45
1 • 1 • 1 1 · · · · · · · · · · · · · · · · ·	46
derived A derived datatype is any datatype that is not predefined.	47
	48

portable A datatype is portable if it is a predefined datatype, or it is derived from
 a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS,
 MPI_TYPE_VECTOR, MPI_TYPE_INDEXED,

⁴ MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY,

⁵ MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable ⁶ because all displacements in the datatype are in terms of extents of one predefined ⁷ datatype. Therefore, if such a datatype fits a data layout in one memory, it will ⁸ fit the corresponding data layout in another memory, if the same declarations were ⁹ used, even if the two systems have different architectures. On the other hand, if a ¹⁰ datatype was constructed using MPI_TYPE_CREATE_HINDEXED,

- ¹¹ MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_HVECTOR or
 - MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different architecture.

equivalent Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

2.5 Data Types

2.5.1 Opaque Objects

²⁵ MPI manages **system memory** that is used for buffering messages and for storing internal ²⁷ representations of various MPI objects such as groups, communicators, datatypes, etc. This ²⁸ memory is not directly accessible to the user, and objects stored there are **opaque**: their ²⁹ size and shape is not visible to the user. Opaque objects are accessed via **handles**, which ³⁰ exist in user space. MPI procedures that operate on opaque objects are passed handle ³¹ arguments to access these objects. In addition to their use by MPI calls for object access, ³² handles can participate in assignments and comparisons.

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

⁴¹ TYPE, BIND(C) :: MPI_Comm

42 INTEGER :: MPI_VAL

43 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.)

Unofficial Draft for Comment Only

12

13

14

15

16

17

18

19

20 21 22

23

 24

40

44

2.5. DATA TYPES

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 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 in C would have been to allow handles to have been an arbitrary, opaque

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 24

type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. In Fortran, the handles are defined such that assignment and comparison are available through the operators of the language or overloaded versions of these operators. (*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.)

²⁵ 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; 31 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.

³⁶ ₃₇ 2.5.3 State

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

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations,
 with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular
 values, which is a proper subrange of the range of values of the corresponding basic type;

Unofficial Draft for Comment Only

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, see Chapter 8. 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:

18 MPI_MAX_PROCESSOR_NAME 19 MPI_MAX_LIBRARY_VERSION_STRING MPI_MAX_ERROR_STRING 2021MPI_MAX_DATAREP_STRING MPI_MAX_INFO_KEY 22 23MPI_MAX_INFO_VAL 24MPI_MAX_OBJECT_NAME 25MPI_MAX_PORT_NAME 26MPI_VERSION MPI_SUBVERSION 27MPI_STATUS_SIZE (Fortran only) 2829MPI_ADDRESS_KIND (Fortran only) MPI_COUNT_KIND (Fortran only) 30 31 MPI_INTEGER_KIND (Fortran only) MPI_OFFSET_KIND (Fortran only) 32 33 MPI_SUBARRAYS_SUPPORTED (Fortran only) 34 MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only) The constants that cannot be used in initialization expressions or assignments in For-35tran are as follows: 36 MPI_BOTTOM 37 MPI_STATUS_IGNORE 38 MPI_STATUSES_IGNORE 39 MPI_ERRCODES_IGNORE 40 MPI_IN_PLACE 41 MPI_ARGV_NULL 42MPI_ARGVS_NULL 43 MPI_UNWEIGHTED MPI_WEIGHTS_EMPTY 44

Advice to implementors. In Fortran the implementation of these special constants 46 may require the use of language constructs that are outside the Fortran standard. 47 Using special values for the constants (e.g., by defining them through PARAMETER 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10 11

12

13 14

15

16

17

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

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include 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 2008 + TR 29113 syntax TYPE(*), DIMENSION(..); for C, we use void *.

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 18.1.1. (End of advice to implementors.)

21 22 23

2.5.6 Absolute Addresses and Relative Address Displacements

 24 Some MPI procedures use *address* arguments that represent an *absolute address* in the call-25ing program, or *relative displacement* arguments that represent differences of two absolute 26addresses. The datatype of such arguments is MPI_Aint in C and INTEGER (KIND= 27MPI_ADDRESS_KIND) in Fortran. These types must have the same width and encode address 28values in the same manner such that address values in one language may be passed directly 29to another language without conversion. There is the MPI constant MPI_BOTTOM to in-30 dicate the start of the address range. For retrieving absolute addresses or any calculation 31 with absolute addresses, one should use the routines and functions provided in Section 4.1.5. 32 Section 4.1.12 provides additional rules for the correct use of absolute addresses. For ex-33 pressions with relative displacements or other usage without absolute addresses, intrinsic 34operators (e.g., +, -, *) can be used. 35

2.5.7 File Offsets

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

44 45

36

37

2.5.8 Counts

⁴⁶ ⁴⁷ As 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

13

14

15 16

17

18

19

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.

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 any prefix of the form "MPI_" and "PMPI_", where any of the letters are either upper or lower case.

2.6.1 Deprecated and Removed Interfaces

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, 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 was declared as 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.

Some of the deprecated constructs are now removed, as documented in Chapter 16. They may still be provided by an implementation for backwards compatibility, but are not

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 24

 1 required.

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

	1 / 1	,	
Deprecated or removed	deprecated	removed	Replacement
construct	since	since	
	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_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT MPI_TYPE_GET_EXTENT
MPI_TYPE_EXTENT MPI_TYPE_UB	MPI-2.0 MPI-2.0	MPI-3.0 MPI-3.0	MPI_TYPE_GET_EXTENT MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI-2.0 MPI-2.0	MPI-3.0 MPI-3.0	MPI_TYPE_GET_EXTENT MPI_TYPE_GET_EXTENT
MPI_LB ¹	MPI-2.0 MPI-2.0		MPI_TYPE_GET_EXTENT MPI_TYPE_CREATE_RESIZED
MPI_LB MPI_UB ¹		MPI-3.0	
MPI_OB MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDL
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 ²	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 ³	MPI-2.0		MPI_COMM_DUP_FN ³
MPI_NULL_COPY_FN ³	MPI-2.0		MPI_COMM_NULL_COPY_FN ³
MPI_NULL_DELETE_FN ³	MPI-2.0		MPI_COMM_NULL_DELETE_FN ³
MPI_Copy_function ²	MPI-2.0		MPI_Comm_copy_attr_function ²
	MPI-2.0		COMM_COPY_ATTR_FUNCTION ³
MPI_Delete_function ²	MPI-2.0		$MPI_Comm_delete_attr_function^2$
DELETE_FUNCTION ³	MPI-2.0		COMM_DELETE_ATTR_FUNCTIO
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 ⁴	-	MPI-3.0	MPI_COMBINER_HVECTOR ⁴
MPI_COMBINER_HINDEXED_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HINDEXED ⁴
MPI_COMBINER_STRUCT_INTEGER ⁴	-	MPI-3.0	$MPI_COMBINER_STRUCT^4$
MPI::	MPI-2.2	MPI-3.0	C language binding
MPI_CANCEL for send requests	MPI-3.2		no direct replacement
MPI_T_ERR_INVALID_ITEM	MPI-3.2		MPI_T_ERR_INVALID_INDEX
¹ Predefined datatype.			
² Callback prototype definition.			
³ Predefined callback routine.			
⁴ Constant.			
Other entries are regular MPI routines.			
Table 2.1: D	eprecated a	nd Remov	ed constructs
2.6.2 Fortran Binding Issues Originally MPI-1 1 provided bindi			1. 1

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

48

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 24

 $\frac{44}{45}$

not declare names, e.g., for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix MPI_. To avoid conflicting with the profiling interface, programs must also avoid subroutines and functions with the prefix PMPI_. This is mandated to avoid possible name collisions.

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

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

Constants representing the maximum length of a string are one smaller in Fortran than in C as discussed in Section 18.2.9.

Handles are represented in Fortran as INTEGERS, or as a BIND(C) derived type with the mpi_f08 module; see Section 2.5.1. 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 18.1.16.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an MPI_ prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare names (identifiers), e.g., for variables, functions, constants, types, or macros, beginning with any prefix of the form MPI_, where any of the letters are either upper or lower case. To support the profiling interface, programs must not declare functions with names beginning with any prefix of the form PMPI_, where any of the letters are either upper or lower case.

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 *.

2.6.4 Functions and Macros

An implementation is allowed to implement MPI_WTIME, PMPI_WTIME, MPI_WTICK, PMPI_WTICK, MPI_AINT_ADD, PMPI_AINT_ADD, MPI_AINT_DIFF, PMPI_AINT_DIFF, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 18.2.4, and no others, as macros in C.

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

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

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD 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.

10 This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of commu-12nication, I/O, and process management is not specified. Unless otherwise stated in the 13 specification of the standard, MPI places no requirements on the result of its interaction 14with external mechanisms that provide similar or equivalent functionality. This includes, 15but is not limited to, interactions with external mechanisms for process control, shared and 16remote memory access, file system access and control, interprocess communication, process 17signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed 19necessary. 20

> Advice to implementations. 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.

Error Handling 2.8

29MPI provides the user with reliable message transmission. A message sent is always received 30 correctly, and the user does not need to check for transmission errors, time-outs, or other 31 error conditions. In other words, MPI does not provide mechanisms for dealing with failures 32 in the communication system. If the MPI implementation is built on an unreliable underly-33 ing mechanism, then it is the job of the implementor of the MPI subsystem to insulate the 34 user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, 35 such failures will be reflected as errors in the relevant communication call. Similarly, MPI 36 itself provides no mechanisms for handling processor failures. 37

Of course, MPI programs may still be erroneous. A **program error** can occur when 38 an MPI call is made with an incorrect argument (non-existing destination in a send oper-39 ation, buffer too small in a receive operation, etc.). This type of error would occur in any 40 implementation. In addition, a resource error may occur when a program exceeds the 41 amount of available system resources (number of pending messages, system buffers, etc.). 42The occurrence of this type of error depends on the amount of available resources in the 43 system and the resource allocation mechanism used; this may differ from system to system. 44 A high-quality implementation will provide generous limits on the important resources so 45as to alleviate the portability problem this represents. 46

In C and Fortran, almost all MPI calls return a code that indicates successful completion 47of the operation. Whenever possible, MPI calls return an error code if an error occurred 48

1

2

3 4

 $\mathbf{5}$ 6

7

8

9

11

18

21

22

23

 24 25

2627

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. On the other hand, some errors may be detected after the associated operation has completed; some errors may not have a communicator, window, or file on which an error may be raised. In such cases, these errors will be raised on the communicator MPI_COMM_SELF.

An example of such a case arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode).

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 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 24

25

26

27 28

29

30

 31

32 33

34

35 36 37

38 39

40

41

42

43

 $\frac{44}{45}$

 $46 \\ 47$

¹ MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is ² 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;
```

7

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();
```

¹³ 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).

```
25
26
```

33

34

35

36 37

38 39

21

2.9.2 Interaction with Signals

²⁷ MPI does not specify the interaction of processes with signals and does not require that MPI ²⁸ be signal safe. The implementation may reserve some signals for its own use. It is required ²⁹ that the implementation document which signals it uses, and it is strongly recommended ³⁰ that it not use SIGALRM, SIGFPE, or SIGIO. Implementations may also prohibit the use of ³¹ 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
- 48

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 7 containing the string message in the memory of process one. The first three parameters 8 of the receive operation specify the location, size and type of the receive buffer. The next 9 three parameters are used for selecting the incoming message. The last parameter is used 10 to return information on the message just received.

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

3.2 Blocking Send and Receive Operations

```
3.2.1 Blocking Send
```

The syntax of the blocking send operation is given below.

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

26	IN	buf	initial address of send buffer (choice)
27 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)

```
    MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
<sup>48</sup> The blocking semantics of this call are described in Section 3.4.
```

Unofficial Draft for Comment Only

17 18

19 20

21

22 23 24

25

35

36

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

 $46 \\ 47$

MPI datatype	C datatype
MPI_CHAR	char
	(treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char
	(treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char
	(treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
	(defined in <stddef.h>)</stddef.h>
	(treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	Tong double _complex
MPI_PACKED	
Table 3.2: Predefined MPI datatypes co	prresponding to C datatypes

INTEGER (KIND=MPI_ADDRESS_KIND), INTEGER (KIND=MPI_OFFSET_KIND), and INTEGER
 (KIND=MPI_COUNT_KIND). This is described in Table 3.3. All predefined datatype handles
 are available in all language bindings. See Sections 18.2.6 and 18.2.10 on page 686 and 694
 for information on interlanguage communication with these types.

If there is an accompanying C++ compiler then the datatypes in Table 3.4 are also
 supported in C and Fortran.

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

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

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

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

3.2.3 Message Envelope

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

source	
destination	
tag	
communicator	

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

The message destination is specified by the **dest** argument.

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

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

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

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

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

Unofficial Draft for Comment Only

1 2 3

5 6

15 16 17

18

19

20

27

28 29

30

 31

32 33

34

35

36 37

38 39

40 41

42

43 44

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 19The syntax of the blocking receive operation is given below. 2021MPI_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) 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(*) 46 INTEGER 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 27wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching 2930 source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=MPI_ANY_TAG in the pattern.

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

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

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

Advice to implementors. Message context and other communicator information can 46be implemented as an additional tag field. It differs from the regular message tag 47in that wild card matching is not allowed on this field, and that value setting for 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8

9

10 11

12

13

14

15

1617

18

19

20

21

22

 24 25

26

28

 31

32

37

38

39

40 41

42

43

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

The use of dest or source=MPI_PROC_NULL to define a "dummy" destination or source in any send or receive call is described in Section 3.11.

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 INTEGER fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR. TYPE(MPI_Status) may contain additional, implementation-specific 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,

³⁰ MPI_ERROR, and TYPE(MPI_Status) are defined to allow conversion between both status ³¹ representations. Conversion routines are provided in Section 18.2.5.

- *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.*)

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

1

 $\mathbf{2}$

3 4

5 6 7

8

32

33

34

35 36

37

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

MPI_GET_COUNT(status, datatype, count)

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

MPI_Get_count(status, datatype, count, ierror)
 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

 $\frac{44}{45}$

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
 described in this section.

12 13

1

2

3

4

5

6

7 8

3.2.6 Passing MPI_STATUS_IGNORE for Status

¹⁴ Every call to MPI_RECV includes a status argument, wherein the system can return details ¹⁵ about the message received. There are also a number of other MPI calls where status ¹⁶ is returned. An object of type MPI_Status is not an MPI opaque object; its structure ¹⁸ is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, ¹⁹ application programs are constructed so that it is unnecessary for them to examine the ²⁰ status fields. In these cases, it is a waste for the user to allocate a status object, and it is ²¹ particularly wasteful for the MPI implementation to fill in fields in this object.

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

²⁴ MPI_STATUS_IGNORE is not a special type of MPI_Status object; rather, it is a special value ²⁶ for the argument. In C one would expect it to be NULL, not the address of a special ²⁷ MPI_Status.

²⁸ MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used every-²⁹ where a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE ³⁰ cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE ³¹ and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or ³² assignment). See Section 2.5.4.

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

⁴¹ MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are not required to have the same ⁴² values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for MPI_{TEST|WAIT}{ALL|SOME} functions set to MPI_STATUS_IGNORE; one either specifies ignoring *all* of the statuses in such a call with MPI_STATUSES_IGNORE, or *none* of them by passing normal statuses in all positions in the array of statuses.

- 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

1 2

5

6 7

8 9

10 11

12

13

1415

16

17

18

19 20

21

22

23

25

 31

32 33

34

35

36

37

38 39

40

41

42

43

4445

4647

```
1
     CALL MPI_COMM_RANK(comm, rank, ierr)
\mathbf{2}
     IF (rank.EQ.0) THEN
3
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
4
     ELSE IF (rank.EQ.1) THEN
5
          CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
6
     END IF
7
          This code is correct if both a and b are real arrays of size \geq 10. (In Fortran, it might be
8
     correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
9
     to an array with ten reals.)
10
11
                      Sender and receiver do not specify matching types.
     Example 3.2
12
13
     CALL MPI_COMM_RANK(comm, rank, ierr)
14
     IF (rank.EQ.0) THEN
15
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
16
     ELSE IF (rank.EQ.1) THEN
17
          CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
18
     END IF
19
          This code is erroneous, since sender and receiver do not provide matching datatype
20
     arguments.
21
22
     Example 3.3
                      Sender and receiver specify communication of untyped values.
23
^{24}
     CALL MPI_COMM_RANK(comm, rank, ierr)
25
     IF (rank.EQ.0) THEN
26
          CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
27
     ELSE IF (rank.EQ.1) THEN
28
          CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
29
     END IF
30
          This code is correct, irrespective of the type and size of a and b (unless this results in
^{31}
     an out of bounds memory access).
32
33
           Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND,
34
           then MPI will send the data stored at contiguous locations, starting from the address
35
           indicated by the buf argument. This may have unexpected results when the data
36
           layout is not as a casual user would expect it to be. For example, some Fortran
37
           compilers implement variables of type CHARACTER as a structure that contains the
38
           character length and a pointer to the actual string. In such an environment, sending
39
           and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have
40
           the anticipated result of transferring the character string. For this reason, the user is
41
           advised to use typed communications whenever possible. (End of advice to users.)
42
43
     Type MPI_CHARACTER
44
45
     The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER,
46
     rather than the entire character string stored in the variable. Fortran variables of type
47
     CHARACTER or substrings are transferred as if they were arrays of characters. This is
48
     illustrated in the example below.
```

```
36
```

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

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

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

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

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

18 Consider the three examples, 3.1-3.3. The first program is correct, assuming that **a** and 19b are REAL arrays of size ≥ 10 . If the sender and receiver execute in different environments, 20then the ten real values that are fetched from the send buffer will be converted to the 21representation for reals on the receiver site before they are stored in the receive buffer. 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.

³⁴ Data representation conversion also applies to the envelope of a message: source, des-³⁵ 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

⁴⁶ MPI requires support for inter-language communication, i.e., if messages are sent by a ⁴⁷ C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined ⁴⁸ in Section 18.2.

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.

According to the definitions in Section 2.4, MPI_BSEND is a blocking procedure and the user can re-use all resources given as arguments, including the message data buffer. It is also a local procedure because it returns immediately without depending on the execution of any MPI procedure in any other MPI process. The same applies to MPI_RSEND because it must not be called before its corresponding receive is not already called.

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

43

44

45

46

47

Advice to users. These are two of the exceptions in which a blocking procedure are local. (End of advice to users.)

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 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 10 communication semantics: a communication does not complete at either end before both 11 processes rendezvous at the communication. A send executed in this mode is non-local. 12

A send that uses the **ready** communication mode may be started *only* if the matching 13 receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-14fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 15required and results in improved performance. The completion of the send operation does 16not depend on the status of a matching receive, and merely indicates that the send buffer 17can be reused. A send operation that uses the ready mode has the same semantics as a 18 standard send operation, or a synchronous send operation; it is merely that the sender 19 provides additional information to the system (namely that a matching receive is already 20posted), that can save some overhead. In a correct program, therefore, a ready send could 21be replaced by a standard send with no effect on the behavior of the program other than 22 performance. 23

Three additional send functions are provided for the three additional communication 24 modes. The communication mode is indicated by a one letter prefix: B for buffered, S for 25synchronous, and R for ready. 26

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

30	IN	buf	initial address of send buffer (choice)
31 32	IN	count	number of elements in send buffer (non-negative integer)
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 40 41	int MPI	_Bsend(const void* buf, in int tag, MPI_Comm co	nt count, MPI_Datatype datatype, int dest, omm)
42	MPI_Bse	nd(buf, count, datatype, d	lest, tag, comm, ierror)
43 44		E(*), DIMENSION(), INTEN	
45		EGER, INTENT(IN) :: count E(MPI_Datatype), INTENT(IN	
46		E(MPI_Comm), INTENT(IN) ::	• -
47	INT	EGER, OPTIONAL, INTENT(OUT	I) :: ierror
48			

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

7

8

9

2728

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*)</type>			1 2
	INTEGER COUNT, DATATYPE, DEST,	TAG, COMM, IERROR	3
	Send in buffered mode.		
I	Send in bunered mode.		5
			6
MPI_	SSEND (buf, count, datatype, dest, t	tag, comm)	7 8
IN	buf	initial address of send buffer (choice)	9
IN	count	number of elements in send buffer (non-negative integer)	10 11
IN	datatype	datatype of each send buffer element (handle)	12
IN	dest	rank of destination (integer)	13
			14 15
IN	tag	message tag (integer)	15
IN	comm	communicator (handle)	17
			18
int		count, MPI_Datatype datatype, int dest,	19
	int tag, MPI_Comm con		20
	Ssend(buf, count, datatype, de	0	21
	TYPE(*), DIMENSION(), INTENT		22 23
	INTEGER, INTENT(IN) :: count,	-	23 24
	TYPE(MPI_Datatype), INTENT(IN) TYPE(MPI_Comm), INTENT(IN) ::	v1	25
	INTEGER, OPTIONAL, INTENT(OUT)		26
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)			27
	SSEND(BUF, CUUNT, DATATYPE, DE <type> BUF(*)</type>	ST, TAG, CUMM, IERRUR)	28
	INTEGER COUNT, DATATYPE, DEST,	TAG, COMM, IERROR	29 30
	Send in synchronous mode.		31
			32
MPI	RSEND (buf, count, datatype, dest,	tag. comm)	33 34
IN	buf	- ,	35
		initial address of send buffer (choice)	36
IN	count	number of elements in send buffer (non-negative integer)	37 38
IN	datatype	datatype of each send buffer element (handle)	39
IN	dest	rank of destination (integer)	40
IN	tag	message tag (integer)	41
IN	comm	communicator (handle)	42
111	comm	communication (nandre)	43 44
int	MPI Rsend(const void* buf int	count, MPI_Datatype datatype, int dest,	45
_110	int tag, MPI_Comm com		46
мрт			47
MP1_	MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) 48		

1 2 3 4 5 6	<pre>TYPE(*), DIMENSION(), INTENT(IN) :: buf INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
7 8 9	MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR</type>
10 11 12 13 14 15 16 17 18 19 20	Send in ready mode. There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is <i>blocking</i> : it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started). In a multithreaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.
21 22 23 24	Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation. It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the
25 26 27 28 29	sender until a matching receive occurs by using the synchronous send mode.A possible communication protocol for the various communication modes is outlined below.<i>ready send</i>: The message is sent as soon as possible.
30 31 32 33	synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.
34 35	standard send: First protocol may be used for short messages, and second protocol for long messages.
36 37 38	<i>buffered send</i> : The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).
39 40	Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.
41 42 43	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.
43 44 45	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.
46 47 48	In a multithreaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (<i>End of advice to implementors.</i>)

3.5 Semantics of Point-to-Point Communication

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

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

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

Example 3.5 An example of non-overtaking messages.

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

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

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

Example 3.6 An example of two, intertwined matching pairs.

 31

1	CALL MPI_COMM_RANK(comm, rank, ierr)
2	IF (rank.EQ.0) THEN
3	CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
4	CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
5	ELSE IF (rank.EQ.1) THEN
6	CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
7	CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
8	END IF

Both processes invoke their first communication call. Since the first send of process zero 10 uses the buffered mode, it must complete, irrespective of the state of process one. Since 11 no matching receive is posted, the message will be copied into buffer space. (If insufficient 12buffer space is available, then the program will fail.) The second send is then invoked. At 13 that point, a matching pair of send and receive operation is enabled, and both operations 14 must complete. Process one next invokes its second receive call, which will be satisfied by 15 the buffered message. Note that process one received the messages in the reverse order they 16 were sent. 17

18

9

MPI makes no guarantee of *fairness* in the handling of communication. Suppose Fairness 19 that a send is posted. Then it is possible that the destination process repeatedly posts a 20receive that matches this send, yet the message is never received, because it is each time 21overtaken by another message, sent from another source. Similarly, suppose that a receive 22 was posted by a multithreaded process. Then it is possible that messages that match this 23receive are repeatedly received, yet the receive is never satisfied, because it is overtaken 24by other receives posted at this node (by other executing threads). It is the programmer's 25responsibility to prevent starvation in such situations. 26

27

28 Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI 2930 call. A quality implementation will use a (small) fixed amount of resources for each pending 31 send in the ready or synchronous mode and for each pending receive. However, buffer space 32 may be consumed to store messages sent in standard mode, and must be consumed to store 33 messages sent in buffered mode, when no matching receive is available. The amount of space 34available for buffering will be much smaller than program data memory on many systems. 35 Then, it will be easy to write programs that overrun available buffer space.

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

41 A buffered send operation that cannot complete because of a lack of buffer space is 42erroneous. When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that 43 cannot complete because of lack of buffer space will merely block, waiting for buffer space 4445to become available or for a matching receive to be posted. This behavior is preferable in 46many situations. Consider a situation where a producer repeatedly produces new values 47and sends them to a consumer. Assume that the producer produces new values faster 48 than the consumer can consume them. If buffered sends are used, then a buffer overflow

will result. Additional synchronization has to be added to the program so as to prevent this from occurring. If standard sends are used, then the producer will be automatically throttled, as its send operations will block when buffer space is unavailable.

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

Example 3.7 An exchange of messages.

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

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

Example 3.8 An errant attempt to exchange messages.

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

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

```
Example 3.9 An exchange that relies on buffering.
```

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

The message sent by each process has to be copied out before the send operation returns ⁴⁵ and the receive operation starts. For the program to complete, it is necessary that at least ⁴⁶ one of the two messages sent be buffered. Thus, this program can succeed only if the ⁴⁷ communication system can buffer at least **count** words of data. ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

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

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

Many programmers prefer to have more leeway and opt to use the "unsafe" program-13 ming style shown in Example 3.9. In such cases, the use of standard sends is likely 14 to provide the best compromise between performance and robustness: quality imple-15mentations will provide sufficient buffering so that "common practice" programs will 16not deadlock. The buffered send mode can be used for programs that require more 17 buffering, or in situations where the programmer wants more control. This mode 18 might also be used for debugging purposes, as buffer overflow conditions are easier to 19 diagnose than deadlock conditions. 20

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

3.6 Buffer Allocation and Usage

46

1

2

3

4

5

6

21

22

23

24

25

26 27

28 29

30

31 32 33 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)

```
34
       IN
                 buffer
                                             initial buffer address (choice)
35
       IN
                 size
                                             buffer size, in bytes (non-negative integer)
36
37
38
     int MPI_Buffer_attach(void* buffer, int size)
39
     MPI_Buffer_attach(buffer, size, ierror)
40
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
41
          INTEGER, INTENT(IN) ::
                                    size
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
43
44
     MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
45
          <type> BUFFER(*)
46
          INTEGER SIZE, IERROR
47
48
```

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, **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 18.1.12).

```
7
MPI_BUFFER_DETACH(buffer_addr, size)
                                                                                        8
           buffer_addr
                                                                                        9
  OUT
                                      initial buffer address (choice)
                                                                                        10
  OUT
                                      buffer size, in bytes (non-negative integer)
           size
                                                                                        11
                                                                                        12
int MPI_Buffer_detach(void* buffer_addr, int* size)
                                                                                        13
                                                                                        14
MPI_Buffer_detach(buffer_addr, size, ierror)
                                                                                        15
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                        16
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
                                                                                        17
    INTEGER, INTENT(OUT) :: size
                                                                                        18
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
                                                                                        19
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
                                                                                        20
    <type> BUFFER_ADDR(*)
                                                                                        21
    INTEGER SIZE, IERROR
                                                                                        22
```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 3.10 Calls to attach and detach buffers.

```
#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value. In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the mpi_f08 module, the address of the buffer is returned as TYPE(C_PTR), see also Example 8.1 about the use of C_PTR pointers. (*End of advice to users.*)

 24

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

56 7

8 9

1

2

3

4

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

MPI must provide as much buffering for outgoing messages as if outgoing message 10 data were buffered by the sending process, in the specified buffer space, using a circular, 11contiguous-space allocation policy. We outline below a model implementation that defines 12this policy. MPI may provide more buffering, and may use a better buffer allocation algo-13 rithm than described below. On the other hand, MPI may signal an error whenever the 14simple buffering allocator described below would run out of space. In particular, if no buffer 1516is explicitly associated with the process, then any buffered send may cause an error.

17MPI 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 implemen-18 19tations.

2021

22

23

24

25

26

27

There is a wide spectrum of possible implementations of buffered com-Rationale. munication: 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.)

2829 30

 31

32

37

38

39 40

41

42

43

44

45

Model Implementation of Buffered Mode 3.6.1

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

33 We assume that a circular queue of pending message entries (PME) is maintained. 34Each entry contains a communication request handle that identifies a pending nonblocking 35 send, a pointer to the next entry and the packed message data. The entries are stored in 36 successive locations in the buffer. Free space is available between the queue tail and the queue head.

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

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- 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 46 arguments used in the MPI_BSEND call, returns an upper bound on the amount 47of space needed to buffer the message data (see Section 4.2). The MPI constant 48

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

Nonblocking Communication 3.7

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 20performance is to use **nonblocking communication**. A nonblocking **send start** call ini-21tiates 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 22to complete the communication, i.e., to verify that the data has been copied out of the send 23buffer. 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. 27A separate **receive complete** call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer 2930 of 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 avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

34Nonblocking 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 35 excepted, can be started whether a matching receive has been posted or not; a nonblocking 36 ready send can be started only if a matching receive is posted. In all cases, the send start 37 call is local: it returns immediately, irrespective of the status of other processes. If the call 38 39 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" 40 41 cases. That is, an MPI implementation should be able to support a large number of pending 42nonblocking 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 4546has 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 4748 may complete, if matched by a nonblocking receive, before the receive complete call occurs.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8

9 10

11

1213

14

1516

17

18

19

 24

2526

28

 31

32 33

43

(It can complete as soon as the sender "knows" the transfer will complete, but before the
 receiver "knows" the transfer will complete.)

³ If the send mode is **buffered** then the message must be buffered if there is no pending ⁴ receive. In this case, the send-complete call is local, and must succeed irrespective of the ⁵ status of a matching receive.

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

10 11 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.

17 Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., 18 the blocking version of buffered send is capable of completing regardless of when a 19matching receive call is made. However, separating the start from the completion 20of these sends still gives some opportunity for optimization within the MPI library. 21For example, starting a buffered send gives an implementation more flexibility in 22 determining if and how the message is buffered. There are also advantages for both 23nonblocking buffered and ready modes when data copying can be done concurrently 24 with computation. 25

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.*)

32 33 34

26

27

28

29

30

 31

3.7.1 Communication Request Objects

³⁵ Nonblocking communications use opaque **request** objects to identify communication oper-³⁶ ations and match the operation that initiates the communication with the operation that ³⁷ terminates it. These are system objects that are accessed via a handle. A request object ³⁸ identifies various properties of a communication operation, such as the send mode, the com-³⁹ munication buffer that is associated with it, its context, the tag and destination arguments ⁴⁰ to be used for a send, or the tag and source arguments to be used for a receive. In addition, ⁴¹ this object stores information about the status of the pending communication operation.

42 43

44

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
 and incomplete) indicates that the call is nonblocking.

48

3.7. NONBLOCKING COMMUNICATION

MPI_ISEI	ND(buf, count, datatype, des	t, 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
			8
IN	tag	message tag (integer)	9
IN	comm	communicator (handle)	10 11
OUT	request	communication request (handle)	12
			13
int MPI_		int count, MPI_Datatype datatype, int dest,	14
	int tag, MPI_Comm	comm, MPI_Request *request)	15
		, dest, tag, comm, request, ierror)	16 17
		TENT(IN), ASYNCHRONOUS :: buf	18
	EGER, INTENT(IN) :: cou E(MPI_Datatype), INTENT	-	19
	E(MPI_Comm), INTENT(IN)		20
	E(MPI_Request), INTENT((21
INTE	EGER, OPTIONAL, INTENT(DUT) :: ierror	22
MPI_ISEN	ND (BUF, COUNT, DATATYPE)	, DEST, TAG, COMM, REQUEST, IERROR)	23 24
	pe> BUF(*)		25
INTE	EGER COUNT, DATATYPE, DI	EST, TAG, COMM, REQUEST, IERROR	26
Star	t a standard mode, nonbloc	king send.	27
	,	5	28
MPL IRS	END(buf, count, datatype, d	est tag comm request)	29 30
		_ ,	31
IN	buf	initial address of send buffer (choice)	32
IN	count	number of elements in send buffer (non-negative inte-	33
		ger)	34
IN	datatype	datatype of each send buffer element (handle)	35
IN	dest	rank of destination (integer)	36 37
IN	tag	message tag (integer)	38
IN	comm	communicator (handle)	39
OUT	request	communication request (handle)	40
			41
int MPI	_Ibsend(const void* buf	, int count, MPI_Datatype datatype, int dest,	42 43
	int tag, MPI_Comm	comm, MPI_Request *request)	43 44
MPI_Ibse	end(buf, count, datatype	e, dest, tag, comm, request, ierror)	45
TYPE	E(*), DIMENSION(), IN	TENT(IN), ASYNCHRONOUS :: buf	46
	EGER, INTENT(IN) :: cou		47
TYPE	E(MPI_Datatype), INTENT	(IN) :: datatype	48

```
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)
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
27
          INTEGER, INTENT(IN) :: count, dest, tag
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          TYPE(MPI_Request), INTENT(OUT) ::
^{31}
                                                 request
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
32
33
     MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
34
          <type> BUF(*)
35
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
36
37
         Start a synchronous mode, nonblocking send.
38
39
40
41
42
43
44
45
46
47
48
```

MPI_IRSE	ND(buf, count, datatype, dest,	tag, comm, request)	1
IN	buf	initial address of send buffer (choice)	2
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			8
	tag	message tag (integer)	9 10
IN	comm	communicator (handle)	10
OUT	request	communication request (handle)	12
int MPI_		nt count, MPI_Datatype datatype, int dest, omm, MPI_Request *request)	13 14 15
	• -	dest, tag, comm, request, ierror) T(IN), ASYNCHRONOUS :: buf	$\frac{16}{17}$
	GER, INTENT(IN) :: count		18
	(MPI_Datatype), INTENT(IN		19
	(MPI_Comm), INTENT(IN) ::		20 21
	(MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT	-	22
			23
	ND(BUF, COUNT, DATATYPE, e> BUF(*)	DEST, TAG, COMM, REQUEST, IERROR)	24
• 1		, TAG, COMM, REQUEST, IERROR	25 26
			20 27
Start	a ready mode nonblocking se	na:	28
			29
MPI_IREC	CV (buf, count, datatype, sourc	e, tag, comm, request)	30
OUT	buf	initial address of receive buffer (choice)	31 32
IN	count	number of elements in receive buffer (non-negative in-teger)	33 34
IN	datatype	datatype of each receive buffer element (handle)	35
IN	source	rank of source or MPI_ANY_SOURCE (integer)	36
IN	tag	message tag or MPI_ANY_TAG (integer)	37 38
IN	comm	communicator (handle)	39
OUT	request	communication request (handle)	40
		- ()	41
int MPI_	Irecv(void* buf, int coun	t, MPI_Datatype datatype, int source,	42 43
	int tag, MPI_Comm co	omm, MPI_Request *request)	43
MPI_Irec	v(buf, count, datatype, s	ource, tag, comm, request, ierror)	45
TYPE	(*), DIMENSION(), ASYNC	HRONOUS :: buf	46
	GER, INTENT(IN) :: count		47
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	48

1 TYPE(MPI_Comm), INTENT(IN) :: comm 2 TYPE(MPI_Request), INTENT(OUT) :: request 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 5<type> BUF(*) 6 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 7 8 Start a nonblocking receive. 9 These calls allocate a communication request object and associate it with the request 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 18.1.10-2018.1.20. (End of advice to users.) 2122233.7.3 Communication Completion 24 The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communica-25tion. The completion of a send operation indicates that the sender is now free to update 26the locations in the send buffer (the send operation itself leaves the content of the send 27buffer unchanged). It does not indicate that the message has been received, rather, it may 28have been buffered by the communication subsystem. However, if a synchronous mode 29 send was used, the completion of the send operation indicates that a matching receive was 30 initiated, and that the message will eventually be received by this matching receive. 31 The completion of a receive operation indicates that the receive buffer contains the 32 received message, the receiver is now free to access it, and that the status object is set. It 33 does not indicate that the matching send operation has completed (but indicates, of course, 34 that the send was initiated). 35 We shall use the following terminology: A null handle is a handle with value 36 MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request 37 is not associated with any ongoing communication (see Section 3.9). A handle is active 38 if it is neither null nor inactive. An **empty** status is a status which is set to return tag 39 = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally 40 configured so that calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and 41 $MPI_GET_ELEMENTS_X$ return count = 0 and $MPI_TEST_CANCELLED$ returns false. We 42set a status variable to empty when the value returned by it is not significant. Status is set 43 in this way so as to prevent errors due to accesses of stale information. 44 The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any 45

⁴⁵ The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any ⁴⁶ of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request ⁴⁷ corresponds to a send call, are undefined, with two exceptions: The error status field will ⁴⁸ 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 that take a single MPI_Status argument, the error code is returned by the function, and the value of the MPI_ERROR field in the MPI_Status argument is undefined (see 3.2.5).

MPI_WAIT	(request, status)				
INOUT	request	request (handle)			
OUT	status	status object (Status)			
int MPI_W	ait(MPI_Request *request,	MPI_Status *status)			
	request, status, ierror)	TT) request			
	MPI_Request), INTENT(INOU MPI_Status) :: status	() :: request			
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR					

A call to MPI_WAIT returns when the operation identified by request is complete. If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated 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.*)

 $\mathbf{2}$

 24

 31

```
1
     MPI_TEST(request, flag, status)
2
       INOUT
                 request
                                             communication request (handle)
3
       OUT
                 flag
                                             true if operation completed (logical)
4
5
       OUT
                                             status object (Status)
                 status
6
7
     int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
8
     MPI_Test(request, flag, status, ierror)
9
          TYPE(MPI_Request), INTENT(INOUT) :: request
10
          LOGICAL, INTENT(OUT) :: flag
11
          TYPE(MPI_Status) :: status
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
15
          LOGICAL FLAG
16
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
17
          A call to MPI_TEST returns flag = true if the operation identified by request is complete.
18
     In such a case, the status object is set to contain information on the completed operation.
19
     If the request is an active persistent request, it is marked as inactive. Any other type of
20
     request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns
21
     flag = false if the operation identified by request is not complete. In this case, the value of
22
     the status object is undefined. MPI_TEST is a local operation.
23
          The return status object for a receive operation carries information that can be accessed
24
     as described in Section 3.2.5. The status object for a send operation carries information
25
     that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).
26
          One is allowed to call MPI_TEST with a null or inactive request argument. In such a
27
     case the operation returns with flag = true and empty status.
28
          The functions MPI_WAIT and MPI_TEST can be used to complete both sends and
29
     receives.
30
^{31}
           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 by using the following operation.

```
MPI_REQUEST_FREE(request)
  INOUT
           request
                                        communication request (handle)
int MPI_Request_free(MPI_Request *request)
MPI_Request_free(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
    MPI_REQUEST_FREE is a local operation. Classes of operations described later in the
standard, such as nonblocking collective and persistent collective (see Chapters 5 and 7),
also use request objects. In the case of nonblocking operations or active persistent requests,
this routine marks the request object for completion and freeing. Ongoing communication,
if any, that is associated with the request will be allowed to continue until it is finished. The
request will be completed and freed only after its associated communication has finished. In
the case of inactive persistent requests it frees the operation, which implies that the user is
allowed to reuse all parameters specified in the initialization of the persistent request. The
use of this routine for generalized requests is described in Section 12.2. In all these cases, the
request is set to MPI_REQUEST_NULL. In the case of nonblocking collective operations and
in the case of persistent collective operations with an active request handle, it is erroneous
to call MPI_REQUEST_FREE.
                  For point-to-point operations, the MPI_REQUEST_FREE mechanism is
     Rationale.
     provided for reasons of performance and convenience on the sending side. (End of
     rationale.)
     Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not
     possible to check for the successful completion of the associated communication with
     calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the
     communication, an error code cannot be returned to the user — such an error must
     be treated as fatal. An active receive request should never be freed as the receiver
     will have no way to verify that the receive has completed and the receive buffer can
```

Example 3.12 An example using MPI_REQUEST_FREE.

be reused. (End of advice to users.)

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank.EQ.0) THEN
D0 i=1, n
CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
CALL MPI_REQUEST_FREE(req, ierr)
CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
CALL MPI_WAIT(req, status, ierr)
```

Unofficial Draft for Comment Only

1

2 3

45

6 7

9

10

11 12

13

14

15

16

17

18

19

20

21

22

23

 24

25

26 27

28

29

30 31

32

33

34

35

36

37

38 39 40

41 42

43 44

 $45 \\ 46$

47

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

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
        CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
        CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank.EQ.1) THEN
        CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
        CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
        CALL MPI_WAIT(r, status, ierr)
END IF
```

```
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)	31 32
INOUT	array_of_requests	array of requests (array of handles)	33
OUT	index	index of handle for operation that completed (integer)	34
OUT	status	status object (Status)	35
001	Status	status object (Status)	36
int MPT W	Jaitany(int count MPI Be	<pre>quest array_of_requests[], int *index,</pre>	37
1110 111 1_1	MPI_Status *status)	quobo array_or_roquobob[], into (index,	38 39
	_		39 40
MPI_Waitany(count, array_of_requests, index, status, ierror)			41
	INTEGER, INTENT(IN) :: count TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)		
	GER, INTENT(OUT) :: inde	· ·	43
	(MPI_Status) :: status		44
INTEC	GER, OPTIONAL, INTENT(OUT)) :: ierror	45
	NV(COUNT ARRAY OF REGUE	STS, INDEX, STATUS, IERROR)	46
		STS, INDEX, STATUS, ILLMON, STATUS_SIZE),	47 48
		······································	-40

 24

1		IERROR	
2 3	Blocks	s until one of the operations a	associated with the active requests in the array has
4	completed.	If more than one operation	n is enabled and can terminate, one is arbitrarily
5			that request in the array and returns in status the
6		· · · · · · · · · · · · · · · · · · ·	e array is indexed from zero in C, and from one in
7			ersistent request, it is marked inactive. Any other
8			equest handle is set to MPI_REQUEST_NULL. ain null or inactive handles. If the list contains no
9			Il entries are null or inactive), then the call returns
10		ly with index = $MPI_UNDEFIN$	
11 12		-	ount, array_of_requests, index, status) has the same
12	effect as t	he execution of $MPI_WAIT(8)$	<pre>karray_of_requests[i], status), where i is the value</pre>
14		- · · · · · · · · · · · · · · · · · · ·	ndex is MPI_UNDEFINED). MPI_WAITANY with an
15	array conta	aining one active entry is equi	ivalent to MPI_WAIT.
16			
17 18	MPI_TEST	ANY(count, array_of_requests	;, index, flag, status)
18	IN	count	list length (non-negative integer)
20	INOUT	array_of_requests	array of requests (array of handles)
21 22 23	OUT	index	index of operation that completed, or MPI_UNDEFINED if none completed (integer)
23 24	OUT	flag	true if one of the operations is complete (logical)
25	OUT	status	status object (Status)
26 27 28	int MPI_T	estany(int count, MPI_Re int *flag, MPI_Statu	quest array_of_requests[], int *index, s *status)
29 30	MPI_Testa	ny(count, array_of_reque	sts, index, flag, status, ierror)
31		ER, INTENT(IN) :: count	
32		-	UT) :: array_of_requests(count)
33		ER, INTENT(OUT) :: inde CAL, INTENT(OUT) :: flag	X
34		[MPI_Status) :: status	
35 36		ER, OPTIONAL, INTENT(OUT) :: ierror
37	MPI_TESTA	NY(COUNT, ARRAY_OF_REQUE	STS, INDEX, FLAG, STATUS, IERROR)
38		CAL FLAG	
39	INTEG	ER COUNT, ARRAY_OF_REQUE	<pre>STS(*), INDEX, STATUS(MPI_STATUS_SIZE),</pre>
40 41		IERROR	
42	Tests	for completion of either one	or none of the operations associated with active
43		-	ag = true, returns in index the index of this request
44			tatus of that operation. If the request is an active
45	persistent	request, it is marked as inact	tive. Any other type of request is deallocated and

persistent request, it is marked as inactive. Any other type of request is deallocated and
the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from
one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns
a value of MPI_UNDEFINED in index and status is undefined.

The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with $\mathsf{flag} = \mathsf{true}$, $\mathsf{index} = \mathsf{MPI}_{\mathsf{UNDEFINED}}$, and an empty status.

If the array of requests contains active handles then the execution of MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_TEST(&array_of_requests[i], flag, status), for i=0, 1, ..., count-1, in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an array containing one active entry is equivalent to MPI_TEST.

MPI_WAITALL(count, array_of_requests, array_of_statuses)

IN	count	lists length (non-negative integer)	13
INOUT	array_of_requests	array of requests (array of handles)	14
OUT	array_of_statuses	array of status objects (array of Status)	15 16
001		array of status objects (array of status)	16
int MPI_W	aitall(int count, MPI_Req	uest array_of_requests[],	18
	MPI_Status array_of_s	statuses[])	19
MPT Waita	11 (count array of reques	ts, array_of_statuses, ierror)	20
	ER, INTENT(IN) :: count	tis, array_or_statuses, rerior/	21
	•	T) :: array_of_requests(count)	22
	MPI_Status) :: array_of_	v i	23
	ER, OPTIONAL, INTENT(OUT)		24
			25 26
		TS, ARRAY_OF_STATUSES, IERROR)	26 27
	ER COUNT, ARRAY_OF_REQUES		27
INIEG	ER ARRAY_OF_STATUSES(MPI_	SIAIUS_SIZE,*), IERRUR	20

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.

40 When one or more of the communications completed by a call to MPI_WAITALL fail, 41 it is desirable to return specific information on each communication. The function 42MPI_WAITALL will return in such case the error code MPI_ERR_IN_STATUS and will set the 43 error field of each status to a specific error code. This code will be MPI_SUCCESS, if the 44specific communication completed; it will be another specific error code, if it failed; or it can 45be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITALL 46will return MPI_SUCCESS if no request had an error, or will return another error code if it 47failed for other reasons (such as invalid arguments). In such cases, it will not update the 48

Unofficial Draft for Comment Only

1

2

3

4

5

6 $\overline{7}$

> 8 9

10 11

12

29

30

31

32

33

34

35

36

37

38

error fields of the statuses. $\mathbf{2}$

> *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.)

```
MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)
```

```
IN
                                            lists length (non-negative integer)
10
                 count
11
       INOUT
                 array_of_requests
                                            array of requests (array of handles)
12
       OUT
                flag
                                            (logical)
13
       OUT
                array_of_statuses
                                            array of status objects (array of Status)
14
15
     int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
16
17
                    MPI_Status array_of_statuses[])
18
     MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
19
         INTEGER, INTENT(IN) :: count
20
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
21
         LOGICAL, INTENT(OUT) :: flag
22
         TYPE(MPI_Status) :: array_of_statuses(*)
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
26
         LOGICAL FLAG
27
         INTEGER COUNT, ARRAY_OF_REQUESTS(*),
```

ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

Returns flag = true if all communications associated with active handles in the array 30 have completed (this includes the case where no handle in the list is active). In this case, each 31 status entry that corresponds to an active request is set to the status of the corresponding 32 operation. Active persistent requests are marked inactive. Requests of any other type are 33 deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. 34 Each status entry that corresponds to a null or inactive handle is set to empty. 35

Otherwise, flag = false is returned, no request is modified and the values of the status 36 entries are undefined. This is a local operation. 37

Errors that occurred during the execution of MPI_TESTALL are handled in the same manner as errors in MPI_WAITALL.

40 41

38

39

28

29

42

43

- 44
- 45
- 46
- 47 48

1

3

4

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)			1
			2
IN	incount	length of array_of_requests (non-negative integer)	3 4
INOUT	array_of_requests	array of requests (array of handles)	4 5
OUT	outcount	number of completed requests (integer)	6
OUT	array_of_indices	array of indices of operations that completed (array of	7
	-	integers)	8 9
OUT	array_of_statuses	array of status objects for operations that completed	10
		(array of Status)	11
			12
int MPI_W	-	_Request array_of_requests[],	13
	int *outcount, int a		14
	MPI_Status array_of_	statuses[])	15
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,			16 17
	array_of_statuses, i		18
	ER, INTENT(IN) :: incou		19
	-	UT) :: array_of_requests(incount)	20
		ount, array_of_indices(*)	21
	MPI_Status) :: array_of		22
INTEG	ER, OPTIONAL, INTENT(OUT) :: lerror	23
MPI_WAITS	OME(INCOUNT, ARRAY_OF_RE	QUESTS, OUTCOUNT, ARRAY_OF_INDICES,	24
	ARRAY_OF_STATUSES, I	ERROR)	25
INTEG		UESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	26
	ARRAY_OF_STATUSES(M	PI_STATUS_SIZE,*), IERROR	27

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.

 $46 \\ 47$

28

29

30

 31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

1 MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses) $\mathbf{2}$ 3 IN incount length of array_of_requests (non-negative integer) 4 INOUT array_of_requests array of requests (array of handles) 56 OUT outcount number of completed requests (integer) 7 OUT array_of_indices array of indices of operations that completed (array of 8 integers) 9 OUT array_of_statuses array of status objects for operations that completed 10 (array of Status) 11 12int MPI_Testsome(int incount, MPI_Request array_of_requests[], 1314int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]) 1516MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, 17array_of_statuses, ierror) 18 INTEGER, INTENT(IN) :: incount 19TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount) 20INTEGER, INTENT(OUT) :: outcount, array_of_indices(*) 21TYPE(MPI_Status) :: array_of_statuses(*) 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2324MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR) 2526INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 27ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR 28Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has 29 completed it returns outcount = 0. If there is no active handle in the list it returns outcount30 = MPI_UNDEFINED. 31 MPI_TESTSOME is a local operation, which returns immediately, whereas 32 MPI_WAITSOME will block until a communication completes, if it was passed a list that 33 contains at least one active handle. Both calls fulfill a **fairness** requirement: If a request 34 for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or 35 MPI_TESTSOME, and a matching send has been posted, then the receive will eventually 36 succeed, unless the send is satisfied by another receive; and similarly for send requests. 37 Errors that occur during the execution of MPI_TESTSOME are handled as for 38 MPI_WAITSOME. 39 40 Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use 41 of MPI_TESTANY. The former returns information on all completed communications, 42with the latter, a new call is required for each communication that completes. 43 A server with multiple clients can use MPI_WAITSOME so as not to starve any client. 44 Clients send messages to the server with service requests. The server calls 45 MPI_WAITSOME with one receive request for each client, and then handles all receives 46 that completed. If a call to MPI_WAITANY is used instead, then one client could starve 47 while requests from another client always sneak in first. (End of advice to users.) 48

Advice to implementors. MPI_TESTSOME should complete as many pending communications as possible. (*End of advice to implementors.*)

Example 3.15 Client-server code (starvation can occur).

```
7
CALL MPI_COMM_SIZE(comm, size, ierr)
                                                                                    8
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                    9
IF(rank .GT. 0) THEN
                              ! client code
                                                                                    10
    DO WHILE(.TRUE.)
                                                                                    11
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
                                                                                    12
       CALL MPI_WAIT(request, status, ierr)
                                                                                    13
    END DO
                                                                                    14
ELSE
              ! rank=0 -- server code
                                                                                    15
       DO i=1. size-1
                                                                                    16
          CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                                                                                    17
                    comm, request_list(i), ierr)
                                                                                    18
       END DO
                                                                                    19
       DO WHILE(.TRUE.)
                                                                                    20
          CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
                                                                                    21
          CALL DO_SERVICE(a(1,index)) ! handle one message
                                                                                    22
          CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
                                                                                    23
                     comm, request_list(index), ierr)
                                                                                    24
       END DO
                                                                                    25
END IF
                                                                                    26
                                                                                    27
Example 3.16
                 Same code, using MPI_WAITSOME.
                                                                                    28
                                                                                    29
                                                                                    30
CALL MPI_COMM_SIZE(comm, size, ierr)
                                                                                    31
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN
                                                                                    32
                             ! client code
                                                                                    33
    DO WHILE(.TRUE.)
                                                                                    34
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
                                                                                    35
                                                                                    36
    END DO
                                                                                    37
ELSE
              ! rank=0 -- server code
                                                                                    38
    DO i=1, size-1
                                                                                    39
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                                                                                    40
                       comm, request_list(i), ierr)
                                                                                    41
    END DO
                                                                                    42
    DO WHILE(.TRUE.)
       CALL MPI_WAITSOME(size, request_list, numdone,
                                                                                    43
                                                                                    44
                         indices, statuses, ierr)
       DO i=1, numdone
                                                                                    45
          CALL DO_SERVICE(a(1, indices(i)))
                                                                                    46
                                                                                    47
          CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                                                                                    48
                        comm, request_list(indices(i)), ierr)
```

1 2

3 4

```
1
              END DO
\mathbf{2}
          END DO
3
     END IF
4
5
      3.7.6
             Non-destructive Test of status
6
      This call is useful for accessing the information associated with a request, without freeing
7
      the request (in case the user is expected to access it later). It allows one to layer libraries
8
      more conveniently, since multiple layers of software may access the same completed request
9
      and extract from it the status information.
10
11
12
      MPI_REQUEST_GET_STATUS(request, flag, status)
13
       IN
                  request
                                              request (handle)
14
15
       OUT
                 flag
                                              boolean flag, same as from MPI_TEST (logical)
16
        OUT
                 status
                                              status object if flag is true (Status)
17
18
      int MPI_Request_get_status(MPI_Request request, int *flag,
19
                     MPI_Status *status)
20
21
      MPI_Request_get_status(request, flag, status, ierror)
22
          TYPE(MPI_Request), INTENT(IN) :: request
23
          LOGICAL, INTENT(OUT) :: flag
^{24}
          TYPE(MPI_Status) :: status
25
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
26
     MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
27
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
28
          LOGICAL FLAG
29
30
```

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

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

3.8 Probe and Cancel

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

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

48

 31

32

33

34

35

36 37

38

Cancelling a send request by calling MPI_CANCEL is deprecated.

3.8.1 Probe

			2
			3
MPI IPRO	BE(source, tag, comm, flag, sta	atus)	4
		,	5
IN	source	rank of source or $MPI_ANY_SOURCE\xspace$ (integer)	6
IN	tag	message tag or MPI_ANY_TAG (integer)	7
IN	comm	communicator (handle)	8
OUT	flag	(logical)	9
	C		10
OUT	status	status object (Status)	11
			12
int MPI_I	probe(int source, int tag	g, MPI_Comm comm, int *flag,	13
	MPI_Status *status)		14
MDT Tomob		atotua iomean)	15
<pre>MPI_Iprobe(source, tag, comm, flag, status, ierror) INTEGER, INTENT(IN) :: source, tag</pre>		16	
TYPE(MPI_Comm), INTENT(IN) :: comm		17	
		comm	18
	AL, INTENT(OUT) :: flag		19
	MPI_Status) :: status		20
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	21
MPI IPROB	E(SOURCE, TAG, COMM, FLAG	. STATUS. IERROR)	22
	AL FLAG		23
INTEG	ER SOURCE, TAG, COMM, STA	TUS(MPI_STATUS_SIZE), IERROR	24
			25
	. –	g, status) returns flag = true if there is a message	26
		he pattern specified by the arguments source, tag,	27
		nessage that would have been received by a call to	28
MPI_RECV	'(, source, tag, comm, status	s) executed at the same point in the program, and	29

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.

returns in status the same value that would have been returned by MPI_RECV(). Otherwise,

MPI_IPROBE is a local procedure since its return does not depend on MPI calls in other MPI processes. It is not associated with an operation. According to the definitions in Section 2.4, it is a blocking procedure although it returns immediately, which is marked with the prefix I.

Advice to users. This is one of the exceptions in which a blocking procedure is local. (End of advice to users.)

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

```
1
     with an arbitrary tag. However, a specific communication context must be provided with
\mathbf{2}
     the comm argument.
3
         It is not necessary to receive a message immediately after it has been probed for, and
4
     the same message may be probed for several times before it is received.
5
          A probe with MPI_PROC_NULL as source returns flag = true, and the status object
6
     returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.11.
7
8
     MPI_PROBE(source, tag, comm, status)
9
10
       IN
                 source
                                             rank of source or MPI_ANY_SOURCE (integer)
11
       IN
                                             message tag or MPI_ANY_TAG (integer)
                 tag
12
       IN
                                             communicator (handle)
                 comm
13
14
       OUT
                                             status object (Status)
                 status
15
16
     int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
17
     MPI_Probe(source, tag, comm, status, ierror)
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_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
24
          INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
25
          MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns
26
     only after a matching message has been found.
27
          The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress:
28
     if a call to MPI_PROBE has been issued by a process, and a send that matches the probe
29
     has been initiated by some process, then the call to MPI_PROBE will return, unless the
30
     message is received by another concurrent receive operation (that is executed by another
^{31}
     thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a
32
     matching message has been issued, then the call to MPI_IPROBE will eventually return flag
33
34
     = true unless the message is received by another concurrent receive operation or matched
     by a concurrent matched probe.
35
36
     Example 3.17
37
          Use blocking probe to wait for an incoming message.
38
39
             CALL MPI_COMM_RANK(comm, rank, ierr)
40
             IF (rank.EQ.0) THEN
41
                  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
42
             ELSE IF (rank.EQ.1) THEN
43
                  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
44
             ELSE IF (rank.EQ.2) THEN
45
                  DO i=1, 2
46
                     CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
47
                                      comm, status, ierr)
48
```

IF (status(MPI_SOURCE) .EQ. 0) THEN CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr) ELSE CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr) END IF END DO END IF

Each message is received with the right type.

Example 3.18 A similar program to the previous example, but now it has a problem.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
       IF (rank.EQ.0) THEN
            CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
       ELSE IF (rank.EQ.1) THEN
            CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.2) THEN
           DO i=1, 2
              CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                              comm, status, ierr)
              IF (status(MPI_SOURCE) .EQ. 0) THEN
100
                   CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                 0, comm, status, ierr)
              ELSE
200
                   CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                                 0, comm, status, ierr)
              END IF
           END DO
       END IF
```

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 48

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

41

42

source **s** with communicator **c** and any tag; in any case, the message probed will be the earliest pending message from source **s** with tag **t** and communicator **c** (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source **s** with tag **t** and communicator **c**, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors.*)

3.8.2 Matching Probe

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

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

```
22
23
```

24

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

```
IN
                                            rank of source or MPI_ANY_SOURCE (integer)
                source
25
26
       IN
                                            message tag or MPI_ANY_TAG (integer)
                tag
27
       IN
                comm
                                            communicator (handle)
28
       OUT
                flag
                                            flag (logical)
29
       OUT
30
                 message
                                            returned message (handle)
^{31}
       OUT
                status
                                            status object (Status)
32
33
     int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
34
                    MPI_Message *message, MPI_Status *status)
35
     MPI_Improbe(source, tag, comm, flag, message, status, ierror)
36
37
          INTEGER, INTENT(IN) :: source, tag
38
          TYPE(MPI_Comm), INTENT(IN) :: comm
39
          LOGICAL, INTENT(OUT) :: flag
40
          TYPE(MPI_Message), INTENT(OUT) ::
                                                 message
41
          TYPE(MPI_Status) :: status
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
43
     MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
44
          INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
45
          LOGICAL FLAG
46
47
          MPI_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is
48
     a message that can be received and that matches the pattern specified by the arguments
```

1

 $\mathbf{2}$

3

4

5

6

7

8

9 10

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.

MPI_IMPROBE is a local procedure. According to the definitions in Section 2.4 and in contrast to MPI_IPROBE, it is a nonblocking procedure because it is the initialization of a matched receive operation.

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 MPI_IMRECV, and the receive operation has started to receive the message sent by the synchronous send.

There is a special predefined message: MPI_MESSAGE_NO_PROC, which is a message which has MPI_PROC_NULL as its source process. The predefined constant MPI_MESSAGE_NULL is the value used for invalid message handles.

A matching probe with MPI_PROC_NULL as source returns flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.11. It is not necessary to call MPI_MRECV or MPI_IMRECV with MPI_MESSAGE_NO_PROC, but it is not erroneous to do so.

Rationale. MPI_MESSAGE_NO_PROC was chosen instead of MPI_MESSAGE_PROC_NULL to avoid possible confusion as another null handle constant. (*End of rationale.*)

MPI_MPROBE(source, tag, comm, message, status)

IN	source	rank of source or MPI_ANY_SOURCE (integer)	37
IN	tag	message tag or $MPI_ANY_TAG\xspace$ (integer)	38
IN	comm	communicator (handle)	39 40
OUT	message	returned message (handle)	41
OUT	status	status object (Status)	42
		• ()	43

int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)

MPI_Mprobe(source, tag, comm, message, status, ierror)
INTEGER, INTENT(IN) :: source, tag

Unofficial Draft for Comment Only

 $\mathbf{2}$

1 2 3 4 5	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Message), INTENT(OUT) :: message TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
6 7		MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR				
8 9 10 11 12 13 14 15 16	 MPI_MPROBE behaves like MPI_IMPROBE except that it is a blocking call that returns only after a matching message has been found. The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progress in the same way as in the case of MPI_PROBE and MPI_IPROBE. According to the definitions in Section 2.4, MPI_MPROBE is incomplete. It is also a non-local procedure. Advice to users. This is one of the exceptions in which incomplete procedures are non-local. (End of advice to users.) 					
17 18	3.8.3 Ma	atched Receives				
19 20 21 22	The functions MPI_MRECV and MPI_IMRECV receive messages that have been previously matched by a matching probe (Section 3.8.2).					
23	MPI_MRE	CV(buf, count, datatype, mes	ssage, status)			
24 25	OUT	buf	initial address of receive buffer (choice)			
26 27	IN	count	number of elements in receive buffer (non-negative in-teger)			
28	IN	datatype	datatype of each receive buffer element (handle)			
29 30	INOUT	message	message (handle)			
31 32	OUT	status	status object (Status)			
33 34	int MPI_M		nt, MPI_Datatype datatype, ge, MPI_Status *status)			
35 36 37 38 39 40 41 42 43	<pre>MPI_Mrecv(buf, count, datatype, message, status, ierror) TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Message), INTENT(INOUT) :: message TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)</pre>					
44 45	01	<type> BUF(*) INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR</type>				
46 47 48			hed by a matching probe operation (Section $3.8.2$).			

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

On return from this function, the message handle is set to MPI_MESSAGE_NULL. All errors that occur during the execution of this operation are handled according to the error handler set for the communicator used in the matching probe call that produced the message handle.

If MPI_MRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the call returns immediately with the status object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a receive from MPI_PROC_NULL was issued (see Section 3.11). A call to MPI_MRECV with MPI_MESSAGE_NULL is erroneous.

According to the definitions in Section 2.4, MPI_MRECV is a blocking procedure and the user can re-use all resources given as arguments, including the message data buffer. It is also a local procedure because it returns immediately without depending on the execution of any MPI procedure in any other MPI process.

Advice to users. This is one of the exceptions in which a blocking procedure is local. (End of advice to users.)

MPI_IMRECV(buf, count, datatype, message, request)			24		
OUT	buf	initial address of receive buffer (choice)	25		
IN	count	number of elements in receive buffer (non-negative in-	26		
		teger)	27 28		
IN	datatype	datatype of each receive buffer element (handle)	29		
INOUT	message	message (handle)	30		
OUT	request	communication request (handle)	31		
			32 33		
int MPI_I	int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype,				
MPI_Message *message, MPI_Request *request)			$\frac{34}{35}$		
MPI_Imrecv(buf, count, datatype, message, request, ierror)			36		
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf			37		
INTEG	ER, INTENT(IN) :: count		38		
TYPE(MPI_Datatype), INTENT(IN) :: datatype					
TYPE(MPI_Message), INTENT(INOUT) :: message			40 41		
TYPE(MPI_Request), INTENT(OUT) :: request					
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	42 43		
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)			44		
<type> BUF(*)</type>					
INTEG	ER COUNT, DATATYPE, MESSA	IGE, REQUEST, IERRUR	46		
MPL I	MRECV is the nonblocking v	ariant of MPI MRECV and starts a nonblocking	47		

MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking ⁴⁷ receive of a matched message. Completion semantics are similar to MPI_IRECV as described ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

1	in Section 3.7.2. On return from this function, the message handle is set to
2	MPI_MESSAGE_NULL.
3	If MPI_IMRECV is called with $MPI_MESSAGE_NO_PROC$ as the message argument, the
4	call returns immediately with a request object which, when completed, will yield a status
5	object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a
6	receive from MPI_PROC_NULL was issued (see Section 3.11). A call to MPI_IMRECV with
7	MPI_MESSAGE_NULL is erroneous.
8	Advice to implementors. If reception of a matched message is started with
9	MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If
10 11	MPI_CANCEL succeeds, the matched message must be found by a subsequent message
11	probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by
12	a subsequent receive operation or cancelled by the sender. See Section 3.8.4 for details
14	about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may
15	fail. (End of advice to implementors.)
16	
17	3.8.4 Cancel
18	
19	
20	MPI_CANCEL(request)
21	IN request communication request (handle)
22	
23	<pre>int MPI_Cancel(MPI_Request *request)</pre>
24	
25	MPI_Cancel(request, ierror)
26	TYPE(MPI_Request), INTENT(IN) :: request
27 28	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28 29	MPI_CANCEL(REQUEST, IERROR)
30	INTEGER REQUEST, IERROR
31	A call to MPI_CANCEL marks for cancellation a pending, nonblocking communica-
32	tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is
33	deprecated. The cancel call is local. It returns immediately, possibly before the communi-
34	
04	cation is actually cancelled. It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or

deprecated. The cancel call is local. It returns immediately, possibly before the communi cation is actually cancelled. It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or
 MPI_TEST (or any of the derived operations) with the cancelled request as argument after
 the call to MPI_CANCEL. If a communication is marked for cancellation, then a MPI_WAIT
 call for that communication is guaranteed to return, irrespective of the activities of other
 processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly
 called in a busy wait loop for a cancelled communication, then MPI_TEST will eventually
 be successful.

⁴¹ MPI_CANCEL can be used to cancel a communication that uses a persistent request (see ⁴² Section 3.9), in the same way it is used for nonpersistent requests. Cancelling a persistent ⁴³ send request by calling MPI_CANCEL is deprecated. A successful cancellation cancels the ⁴⁴ active communication, but not the request itself. After the call to MPI_CANCEL and the ⁴⁵ subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be ⁴⁶ activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message. Cancelling a buffered send request by calling MPI_CANCEL is deprecated.

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

If the operation has been 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)		
IN	status	status object (Status)
OUT	flag	(logical)
		Status *status, int *flag)
<pre>MPI_Test_cancelled(status, flag, ierror) TYPE(MPI_Status), INTENT(IN) :: status</pre>		
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,

Unofficial Draft for Comment Only

 $\mathbf{2}$

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 (with the exception of the buffer contents) 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 operations. In the case of point-to-point communication, 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 point-to-point request be received by a receive operation using a persistent point-to-point request, or vice versa.

There are also collective communication persistent operations defined in Section 5.13 and Section 7.8. The remainder of this section covers the point-to-point persistent initialization operations and the start routines, which are used for both point-to-point and collective persistent communication.

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

24 25 26

39

23

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

		· · · ·	,
27 28	IN	buf	initial address of send buffer (choice)
29	IN	count	number of elements sent (non-negative integer)
30	IN	datatype	type of each element (handle)
31 32	IN	dest	rank of destination (integer)
33	IN	tag	message tag (integer)
34	IN	comm	communicator (handle)
35 36	OUT	request	communication request (handle)
37			
38	int MPI_	<pre>Send_init(const void* buf</pre>	, int count, MPI_Datatype datatype,

int dest, int tag, MPI_Comm comm, MPI_Request *request)

```
40
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
42
         INTEGER, INTENT(IN) :: count, dest, tag
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Comm), INTENT(IN) ::
                                         comm
45
         TYPE(MPI_Request), INTENT(OUT) ::
                                             request
46
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
47
     MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
48
```

Unofficial Draft for Comment Only

1

2

3 4

5 6

7

8

9

10

11

12

13

14

15

16

	e> BUF(*) GER COUNT, DATATYPE, DEST	, TAG, COMM, REQUEST, IERROR	1 2
	Creates a persistent communication request for a standard mode send operation, and		
	t all the arguments of a send		4
	0	1	5 6
MPI BSF	ND_INIT(buf, count, datatype,	dest tag comm request)	7
IN IN	buf	·	8
		initial address of send buffer (choice)	9
IN	count	number of elements sent (non-negative integer)	10 11
IN	datatype	type of each element (handle)	12
IN	dest	rank of destination (integer)	13
IN	tag	message tag (integer)	14
IN	comm	communicator (handle)	15 16
OUT	request	communication request (handle)	17
			18
int MPI_		(DI Comm count, MPI_Datatype datatype,	19
	C C	<pre>MPI_Comm comm, MPI_Request *request)</pre>	20 21
	-	rpe, dest, tag, comm, request, ierror)	22
	GER, INTENT(IN) :: count	T(IN), ASYNCHRONOUS :: buf	23
	(MPI_Datatype), INTENT(IN	-	24
TYPE	TYPE(MPI_Comm), INTENT(IN) :: comm		
	(MPI_Request), INTENT(OUT	-	26 27
LNTE	GER, OPTIONAL, INTENT(OUT) :: ierror	28
	MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)		
• -	e> BUF(*)	', TAG, COMM, REQUEST, IERROR	30
			31 32
Crea	tes a persistent communication	n request for a buffered mode send.	33
			34
MPI_SSE	ND_INIT(buf, count, datatype,	dest, tag, comm, request)	35
IN	buf	initial address of send buffer (choice)	36 37
IN	count	number of elements sent (non-negative integer)	38
IN	datatype	type of each element (handle)	39
IN	dest	rank of destination (integer)	40
IN	tag	message tag (integer)	41 42
IN	comm	communicator (handle)	43
OUT	request	communication request (handle)	44
001	request	communication request (nanue)	45
int MPI_	int MPI_Ssend_init(const void* buf, int count, MPI_Datatype datatype, 46		
_	int dest, int tag, MPI_Comm comm, MPI_Request *request)		

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

MPI_REC	CV_INIT(buf, count, d	atatype, source, tag, comm, request)	1
OUT	buf	initial address of receive buffer (choice)	2 3
IN	count	number of elements received (non-negative integer)	4
IN	datatype	type of each element (handle)	5
IN	source	rank of source or MPI_ANY_SOURCE (integer)	6
IN	tag	message tag or MPI_ANY_TAG (integer)	7 8
IN	comm	communicator (handle)	9
OUT	request	communication request (handle)	10
001	request	communication request (namule)	11
int MPI		uf, int count, MPI_Datatype datatype, int source, _Comm comm, MPI_Request *request)	12 13 14
TYPI INTI TYPI TYPI TYPI	E(*), DIMENSION(EGER, INTENT(IN) : E(MPI_Datatype), I E(MPI_Comm), INTEN	TENT(OUT) :: request	15 16 17 18 19 20 21
<ty]< td=""><td>pe> BUF(*)</td><td>DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) PE, SOURCE, TAG, COMM, REQUEST, IERROR</td><td>22 23 24 25</td></ty]<>	pe> BUF(*)	DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) PE, SOURCE, TAG, COMM, REQUEST, IERROR	22 23 24 25
is marked the argun A per munication A co	as OUT because the ment to MPI_RECV_I ersistent communication is attached to the	on request is inactive after it was created — no active com-	26 27 28 29 30 31 32 33
MPI_STA	ART(request)		$34 \\ 35$
INOUT	request	communication request (handle)	36 37
int MPI	_Start(MPI_Request	*request)	38 39
TYPI	rt(request, ierror E(MPI_Request), IN EGER, OPTIONAL, IN		40 41 42
	RT(REQUEST, IERROR EGER REQUEST, IERR		43 44 45
	e , . ,	s a handle returned by one of the previous five calls. The nactive. The request becomes active once the call is made.	$\frac{46}{47}$

Unofficial Draft for Comment Only

1 If the request is for a send with ready mode, then a matching receive should be posted $\mathbf{2}$ before the call is made. The communication buffer should not be modified after the call, 3 and until the operation completes. 4 The call is local, with similar semantics to the nonblocking communication operations $\mathbf{5}$ described in Section 3.7. That is, a call to MPI_START with a request created by 6 MPI_SEND_INIT starts a communication in the same manner as a call to MPI_ISEND; a 7call to MPI_START with a request created by MPI_BSEND_INIT starts a communication 8 in the same manner as a call to MPI_IBSEND; and so on.

MPI_STARTALL(count, array_of_requests) 11

IN count list length (non-negative integer) INOUT array_of_requests array of requests (array of handle) int MPI_Startall(int count, MPI_Request array_of_requests[]) MPI_Startall(count, array_of_requests, ierror) INTEGER, INTENT(IN) :: count 19TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to

26MPI_START (&array_of_requests[i]), executed for $i=0, \ldots, count-1$, in some arbitrary order. 27A communication started with a call to MPI_START or MPI_STARTALL is completed 28by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Sec-29tion 3.7.5. The request becomes inactive after successful completion of such call. The re-30 quest is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL 31 call. 32

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

33 The call to MPI_REQUEST_FREE can occur at any point in the program after the per-34sistent request was created. However, the request will be deallocated only after it becomes 35 inactive. Active receive requests should not be freed. Otherwise, it will not be possible to 36 check that the receive has completed. Collective operation requests (defined in Section 5.1237 and Section 7.7 for nonblocking collective operations, and Section 5.13 and Section 7.8 for 38 persistent collective operations) must not be freed while active. It is preferable, in general, 39 to free requests when they are inactive. If this rule is followed, then the functions described 40in this section will be invoked in a sequence of the form,

41 42

Create (Start Complete)* Free

43where * indicates zero or more repetitions. If the same communication object is used in 44several concurrent threads, it is the user's responsibility to coordinate calls so that the 45correct sequence is obeyed.

46A send operation initiated with MPI_START can be matched with any receive operation 47and, likewise, a receive operation initiated with MPI_START can receive messages generated 48by any send operation.

9 10

12

13

1415

1617

18

20

21

22

23 24

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 18.1.10-18.1.20. (End of advice to users.)

3.10 Send-Receive

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

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

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype,

	source, recvtag, comm, st	atus)	24
IN	sendbuf	initial address of send buffer (choice)	25
IN	sendcount	number of elements in send buffer (non-negative inte-	26 27
		ger)	28
IN	sendtype	type of elements in send buffer (handle)	29
IN	dest	rank of destination (integer)	$30 \\ 31$
IN	sendtag	send tag (integer)	32
OUT	recvbuf	initial address of receive buffer (choice)	33
IN	recvcount	number of elements in receive buffer (non-negative in- teger)	34 35
IN	recvtype	type of elements in receive buffer (handle)	36
	Тестуре	×-	37
IN	source	rank of source or MPI_ANY_SOURCE (integer)	38 39
IN	recvtag	receive tag or MPI_ANY_TAG (integer)	39 40
IN	comm	communicator (handle)	41
OUT	status	status object (Status)	42
			43
			44

44int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, 45int dest, int sendtag, void *recvbuf, int recvcount, 46MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, 47MPI_Status *status) 48

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

2122

```
1
     MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
\mathbf{2}
                    recvcount, recvtype, source, recvtag, comm, status, ierror)
3
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
4
          TYPE(*), DIMENSION(..) :: recvbuf
5
          INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
6
                     recvtag
7
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          TYPE(MPI_Status) :: status
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
12
                    RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
13
          <type> SENDBUF(*), RECVBUF(*)
14
          INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
15
                     SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
16
17
         Execute a blocking send and receive operation. Both send and receive use the same
18
     communicator, but possibly different tags. The send buffer and receive buffers must be
19
     disjoint, and may have different lengths and datatypes.
20
          The semantics of a send-receive operation is what would be obtained if the caller forked
21
     two concurrent threads, one to execute the send, and one to execute the receive, followed
22
     by a join of these two threads.
23
24
     MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recytag, comm,
25
                    status)
26
27
       INOUT
                 buf
                                             initial address of send and receive buffer (choice)
28
       IN
                                             number of elements in send and receive buffer (non-
                 count
29
                                             negative integer)
30
       IN
                 datatype
                                             type of elements in send and receive buffer (handle)
^{31}
32
       IN
                 dest
                                             rank of destination (integer)
33
       IN
                 sendtag
                                            send message tag (integer)
34
                                             rank of source or MPI_ANY_SOURCE (integer)
       IN
                 source
35
36
       IN
                                             receive message tag or MPI_ANY_TAG (integer)
                 recvtag
37
       IN
                 comm
                                             communicator (handle)
38
       OUT
                 status
                                             status object (Status)
39
40
41
     int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
42
                    int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
43
                    MPI_Status *status)
44
     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
45
                    comm, status, ierror)
46
          TYPE(*), DIMENSION(..) :: buf
47
          INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
48
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

Advice to implementors. Additional intermediate buffering is needed for the "replace" variant. (End of advice to implementors.)

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 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 succeeds and returns as soon as possible, and the status object returns source = MPI_PROC_NULL as source returns flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0.

 $\mathbf{2}$

Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 and in Section 3.3. 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 communications have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shapes and sizes. It is not assumed that the MPI library is cognizant of 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.

1	A general datatype is an opaque object that specifies two things:
•	A sequence of basic datatypes
•	A sequence of integer (byte) displacements

7 8 9

15

16

17

18 19

2021

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

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

6 7

1

 $\mathbf{2}$

3

4

5

8 9

10

12

13

14

15

18

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

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

11

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

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

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

19We can use a handle to a general datatype as an argument in a send or receive operation, 20instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will 21use the send buffer defined by the base address **buf** and the general datatype associated 22 with datatype; it will generate a message with the type signature determined by the datatype 23argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base 24 address **buf** and the general datatype associated with **datatype**. 25

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

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

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

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

then

38 39 40

32

33

3435

36

37

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

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

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

41 42

43If $type_j$ requires alignment to a byte address that is a multiple of k_j , then ϵ is the least 44non-negative increment needed to round extent(Typemap) to the next multiple of $\max_i k_i$. 45In Fortran, it is implementation dependent whether the MPI implementation computes 46the alignments k_i according to the alignments used by the compiler in common blocks, 47SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE 48nor BIND(C). The complete definition of extent is given by Equation 4.1 Section 4.1.

Unofficial Draft for Comment Only

Let

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

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, SEQUENCE derived types, or BIND(C) derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI_TYPE_CREATE_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 4.1.6 and in Section 18.1.15. (End of rationale.)

4.1.1 Type Constructors with Explicit Addresses

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

4.1.2 Datatype Constructors

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

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (non-negative integer)	34
	- - +		35
IN	oldtype	old datatype (handle)	36
OUT	newtype	new datatype (handle)	37
			38
int MPI_Ty	pe_contiguous(int count,	MPI_Datatype oldtype,	39
	MPI_Datatype *newtype		40
			41
• -	contiguous(count, oldtype	, newtype, ierror)	42
	R, INTENT(IN) :: count		43
TYPE(M	<pre>IPI_Datatype), INTENT(IN)</pre>	:: oldtype	44
TYPE(M	<pre>IPI_Datatype), INTENT(OUT</pre>) :: newtype	45
INTEGE	R, OPTIONAL, INTENT(OUT)	:: ierror	46
MPI_TYPE_C	CONTIGUOUS(COUNT, OLDTYPE	, NEWTYPE, IERROR)	47
INTEGE	R COUNT, OLDTYPE, NEWTYP	E, IERROR	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

1 2			by concatenating count copies of <i>extent</i> as the size of the concatenated copies.
3 4 5	Example 4.2 Let oldtype have type map $\{(double, 0), (char, 8)\}$, with extent 16, and let count = 3. The type map of the datatype returned by newtype is		
6	{(doi	(uble, 0), (char, 8), (double, 16)	$), (char, 24), (double, 32), (char, 40) \};$
7 8	i.e., alterna	ating double and char elemen	ts, with displacements $0, 8, 16, 24, 32, 40$.
9	In gen	eral, assume that the type ma	ap of oldtype is
10	-	$(be_0, disp_0), \dots, (type_{n-1}, disp_n)$	
11 12			e map with $count \cdot n$ entries defined by:
13), $(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex),$
14			$\dots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$
15 16	$\ldots,(\iota y)$	$pe_0, aisp_0 + ex \cdot (\operatorname{count} - 1)),$	$\ldots, (lype_{n-1}, disp_{n-1} + ex \cdot (\text{count} - 1))\}.$
17 18 19 20 21	cation of a obtained b	a datatype into locations that	OR is a more general constructor that allows repli- t consist of equally spaced blocks. Each block is mber of copies of the old datatype. The spacing at of the old datatype.
22 23	MPI_TYPE	E_VECTOR(count, blocklength,	, stride, oldtype, newtype)
23 24	IN	count	number of blocks (non-negative integer)
25 26	IN	blocklength	number of elements in each block (non-negative inte- ger)
27 28 29	IN	stride	number of elements between start of each block (integer)
30	IN	oldtype	old datatype (handle)
31	OUT	newtype	new datatype (handle)
32 33 34 35	int MPI_T	• 1	t blocklength, int stride, , MPI_Datatype *newtype)
36 37 38 39 40	<pre>MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror) INTEGER, INTENT(IN) :: count, blocklength, stride TYPE(MPI_Datatype), INTENT(IN) :: oldtype TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>		
41 42 43			H, STRIDE, OLDTYPE, NEWTYPE, IERROR) TRIDE, OLDTYPE, NEWTYPE, IERROR
44 45 46 47	-	A call to MPI_TYPE_VECTOR	dtype has type map $\{(double, 0), (char, 8)\}$, with R(2, 3, 4, oldtype, newtype) will create the datatype
48	{(doi	uble, 0), (char, 8), (double, 16)), (char, 24), (double, 32), (char, 40),

"heterogeneous").

(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104) }. 1 2 That is, two blocks with three copies each of the old type, with a stride of 4 elements $(4 \cdot 16)$ 3 bytes) between the the start of each block. 4 5**Example 4.4** A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the 6 datatype, 7 $\{(double, 0), (char, 8), (double, -32), (char, -24), (double, -64), (char, -56)\}.$ 8 9 In general, assume that oldtype has type map, 10 $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ 11 12with extent ex. Let bl be the blocklength. The newly created datatype has a type map with 13 $count \cdot bl \cdot n$ entries: 14 $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \}$ 1516 $(type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,$ 17 $(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$ 18 19 $(type_0, disp_0 + \mathsf{stride} \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot ex), \ldots,$ 2021 $(type_0, disp_0 + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), \dots,$ 22 $(type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \ldots,$ 23 24 $(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \ldots,$ 2526 $(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \ldots,$ 27 $(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$ 2829A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to 30 MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, 31 count, n, oldtype, newtype), n arbitrary. 32 33 **Hvector** The function MPI_TYPE_CREATE_HVECTOR is identical to 34 MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The 35 use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for 36

			38
MPI_TYPE	CREATE_HVECTOR(count,	blocklength, stride, oldtype, newtype)	39
IN	count	number of blocks (non-negative integer)	40
			41
IN	blocklength	number of elements in each block (non-negative inte-	42
		$\operatorname{ger})$	43
IN	stride	number of bytes between start of each block (integer)	44
IN	oldtype	old datatype (handle)	45
	21		46
OUT	newtype	new datatype (handle)	47
			48

Unofficial Draft for Comment Only

```
1
      int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
\mathbf{2}
                       MPI_Datatype oldtype, MPI_Datatype *newtype)
3
      MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
4
                        ierror)
5
           INTEGER, INTENT(IN) :: count, blocklength
6
           INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
7
           TYPE(MPI_Datatype), INTENT(IN) :: oldtype
8
           TYPE(MPI_Datatype), INTENT(OUT) :: newtype
9
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
      MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
12
                        IERROR)
13
           INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
14
           INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
15
           Assume that oldtype has type map,
16
17
            \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\
18
      with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
19
      count \cdot bl \cdot n entries:
20
21
            \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), 
22
23
            (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,
24
25
            (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
26
27
            (type_0, disp_0 + \mathsf{stride}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \ldots,
28
29
            (type_0, disp_0 + stride + (bl - 1) \cdot ex), \ldots,
30
31
            (type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex), \ldots,
32
            (type_0, disp_0 + stride \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)), \ldots,
33
34
            (type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex), \ldots,
35
36
            (type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.
37
38
      Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a
39
      sequence of blocks (each block is a concatenation of the old datatype), where each block
40
      can contain a different number of copies and have a different displacement. All block
41
      displacements are multiples of the old type extent.
42
43
44
45
46
47
```

MPI_TY	PE_INDEXED(count, array_of_ newtype)	_blocklengths, array_of_displacements, oldtype,	1 2
IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non- negative integer)	3 4 5 6
IN	array_of_blocklengths	number of elements per block (array of non-negative integers)	7 8
IN	array_of_displacements	displacement for each block, in multiples of oldtype extent (array of integer)	9 10
IN	oldtype	old datatype (handle)	11
OUT	newtype	new datatype (handle)	12 13
int MPI	• •	<pre>const int array_of_blocklengths[], displacements[], MPI_Datatype oldtype, pe)</pre>	14 15 16 17
INT TYP TYP	oldtype, newtype, i	t, array_of_blocklengths(count), ents(count) N) :: oldtype UT) :: newtype	18 19 20 21 22 23 24 25
	OLDTYPE, NEWTYPE, J	KLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),	26 27 28 29 30
and let [oldtype have type map $\{(dout)$	(ble, 0), (char, 8), with extent 16. Let $B = (3, 1)YPE_INDEXED(2, B, D, oldtype, newtype) returns a$	31 32 33 34 35
{(d	louble, 64), (char, 72), (double)	(a, 80), (char, 88), (double, 96), (char, 104),	$36 \\ 37$
(dc	$\texttt{puble}, 0), (\texttt{char}, 8) \}.$		38
displacer		carting at displacement 64, and one copy starting at as type map,	39 40 41 42
0	$ype_0, disp_0), \ldots, (type_{n-1}, disp_n)$		43
			44 45
		blocklengths argument and D be the newly created datatype has $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$ entries:	46
		$ppe_{n-1}, disp_{n-1} + D[0] \cdot ex), \dots,$	47 48

Unofficial Draft for Comment Only

1	$(type_0$	$b, disp_0 + (D[0] + B[0] - 1) \cdot es$	$x),\ldots,$
2 3	$(type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), \dots,$		
4 5	$(type_0, disp_0 + D[count-1] \cdot ex), \dots, (type_{n-1}, disp_{n-1} + D[count-1] \cdot ex), \dots,$		
6 7	$(type_0$	$\mathbf{D}, disp_0 + (D[count-1] + B[cou])$	$nt-1]-1)\cdot ex),\ldots,$
8	$(type_r$	$a_{n-1}, disp_{n-1} + (D[count-1] + E)$	$B[count-1] - 1) \cdot ex)\}.$
9 10 11			t, blocklength, stride, oldtype, newtype) is equivalent , B, D, oldtype, newtype) where
12 13	D[j] =	$j \cdot stride, \ j = 0, \dots, count - $	1,
14	and		
15 16	B[j] =	blocklength, $j = 0, \ldots, \text{count}$	z — 1.
17 18 19 20 21	MPI_TYPE		REATE_HINDEXED is identical to a displacements in array_of_displacements are spec- f the oldtype extent.
22 23	MPI_TYPE	_CREATE_HINDEXED(count, oldtype, newtype)	array_of_blocklengths, array_of_displacements,
24 25 26 27	IN	count	<pre>number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non- negative integer)</pre>
28 29	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
30	IN	array_of_displacements	byte displacement of each block (array of integer)
31 32	IN	oldtype	old datatype (handle)
33	OUT	newtype	new datatype (handle)
34 35 36 37	int MPI_Ty	•	<pre>count, const int array_of_blocklengths[], _of_displacements[], MPI_Datatype oldtype, e)</pre>
38 39 40 41 42 43 44 45 46	INTEGE INTEGE TYPE(N TYPE(N	· -	<pre>ts, oldtype, newtype, ierror) , array_of_blocklengths(count) , INTENT(IN) :: nts(count)) :: oldtype T) :: newtype</pre>
47 48	MPI_TYPE_C	CREATE_HINDEXED(COUNT, AF ARRAY_OF_DISPLACEMEN	RRAY_OF_BLOCKLENGTHS, IS, OLDTYPE, NEWTYPE, IERROR)

1 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR 2 INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*) Assume that oldtype has type map, 4 5 $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ 6 with extent ex. Let B be the array_of_blocklengths argument and D be the 7 array_of_displacements argument. The newly created datatype has a type map with $n \cdot$ 8 $\sum_{i=0}^{\text{count}-1} B[i]$ entries: 9 10 $\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, disp_{n-1} + D[0]), \dots, \}$ 11 12 $(type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex), \dots,$ 13 14 $(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots,$ 15 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}]), \dots, (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}]), \dots,$ 1617 18 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,$ 19 $(type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$ 2021Indexed_block This function is the same as MPI_TYPE_INDEXED except that the block-22 23length is the same for all blocks. There are many codes using indirect addressing arising 24 from unstructured grids where the blocksize is always 1 (gather/scatter). The following 25convenience function allows for constant blocksize and arbitrary displacements. 2627MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, 28 newtype) 2930 length of array of displacements (non-negative integer) IN count 31 blocklength IN size of block (non-negative integer) 32 array_of_displacements IN array of displacements (array of integer) 33 34 IN oldtype old datatype (handle) 35 OUT newtype new datatype (handle) 36 37 int MPI_Type_create_indexed_block(int count, int blocklength, 38 const int array_of_displacements[], MPI_Datatype oldtype, 39 MPI_Datatype *newtype) 40 41 MPI_Type_create_indexed_block(count, blocklength, array_of_displacements, 42oldtype, newtype, ierror) 43 INTEGER, INTENT(IN) :: count, blocklength, 44 array_of_displacements(count) 45TYPE(MPI_Datatype), INTENT(IN) :: oldtype

TYPE(MPI_Datatype), INTENT(OUT) :: newtype

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

46

47

1 2	MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)			
3 4 5	INTE	EGER COUNT, BLOCKLENGTH, NEWTYPE, IERROR	ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,	
6				
7			YPE_CREATE_HINDEXED_BLOCK is identical to	
8			CK, except that block displacements in bytes, rather than in multiples of the oldtype extent.	
9 10	anay_01_	displacements are specified in	bytes, rather than in multiples of the Outype extent.	
11				
12 13		oldtype, newtype)	<pre>OCK(count, blocklength, array_of_displacements,</pre>	
14	IN	count	$length \ of \ array \ of \ displacements \ (non-negative \ integer)$	
15	IN	blocklength	size of block (non-negative integer)	
16 17	IN	array_of_displacements	byte displacement of each block (array of integer)	
18	IN	oldtype	old datatype (handle)	
19	OUT	newtype	new datatype (handle)	
20				
21 22	int MPI_		ock(int count, int blocklength,	
23		const MP1_Aint arra MPI_Datatype *newty	<pre>ay_of_displacements[], MPI_Datatype oldtype,</pre>	
24		• • •	-	
25 26	<pre>MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements, oldtype, newtype, ierror)</pre>			
20	INTEGER, INTENT(IN) :: count, blocklength			
28				
29				
30				
31 32				
33			COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	
34		OLDTYPE, NEWTYPE, I		
35	INTE		OLDTYPE, NEWTYPE, IERROR	
36 37	INTE	EGER(KIND=MPI_ADDRESS_KIN	<pre>iD) ARRAY_OF_DISPLACEMENTS(*)</pre>	
38				
39	Struct N	<pre>//PI_TYPE_CREATE_STRUCT</pre>	T is the most general type constructor. It further	
40	0		DEXED in that it allows each block to consist of repli-	
41 42	cations of	f different datatypes.		
42 43				
44				
45				
46				
47 48				
10				

MPI_TYF	E_CREATE_STRUCT(count, array_of_types, newtyp	anay_or_blocklengths, anay_or_displacements,	$\frac{1}{2}$
IN	count	number of blocks (non-negative integer) — also num- ber of entries in arrays array_of_types, array of displacements and array of blocklengths	3 4 5 6
IN	array_of_blocklength	number of elements in each block (array of non-negative	7 8
IN	array_of_displacements	byte displacement of each block (array of integer)	9
IN	array_of_types	type of elements in each block (array of handles to datatype objects)	10 11 12
OUT	newtype	new datatype (handle)	13 14
int MPI_	const MPI_Aint arra	ount, const int array_of_blocklengths[], y_of_displacements[], array_of_types[], MPI_Datatype *newtype)	15 16 17
INTE INTE TYPE TYPE INTE MPI_TYPE INTE	EGER, INTENT(IN) :: coun EGER(KIND=MPI_ADDRESS_KIN array_of_displacem E(MPI_Datatype), INTENT(I E(MPI_Datatype), INTENT(O EGER, OPTIONAL, INTENT(OU E_CREATE_STRUCT(COUNT, AR ARRAY_OF_DISPLACEME EGER COUNT, ARRAY_OF_BLOC IERROR	<pre>ray_of_blocklengths, ents, array_of_types, newtype, ierror) t, array_of_blocklengths(count) D), INTENT(IN) :: ents(count) N) :: array_of_types(count) UT) :: newtype T) :: ierror RAY_OF_BLOCKLENGTHS, ENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR) KLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE, D) ARRAY_OF_DISPLACEMENTS(*)</pre>	18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Example	e 4.6 Let type1 have type ma	ap,	33
{(d	$\texttt{ouble}, 0), (\texttt{char}, 8)\},$		34 35
	all to MPI_TYPE_CREATE_S	0, 16, 26), and $T = (MPI_FLOAT, type1, MPI_CHAR)$. $\Gamma RUCT(3, B, D, T, newtype)$ returns a datatype with	36 37 38 39
$\{(f)\}$	loat, 0), (float, 4), (double, 1)	(16), (char, 24), (char, 26), (char, 27), (char, 28).	$40 \\ 41$
16, follow four byte	red by three copies of MPI_CH s.)	rting at 0, followed by one copy of type1 starting at AR, starting at 26. (We assume that a float occupies ypes argument, where T[i] is a handle to,	42 43 44 45
type	$emap_i = \{(type_0^i, disp_0^i), \dots, ($		$46 \\ 47$

3)) 1

1 2 3	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:				
4 5	$\{(type_0^0, disp_0^0 + D[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0]), \dots, \}$				
6 7	(typ)	$be_0^0, disp_0^0 + D[0] + (B[0] - 1) \cdot de_0^0$	$ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0]-1) \cdot ex_0), \dots,$		
8	(ty)	$be_0^{C-1}, disp_0^{C-1} + D[c-1]), \dots, (t_2)$	$ype_{nc_{1}-1}^{C-1}, disp_{nc_{1}-1}^{C-1} + D[c-1]), \dots,$		
9 10					
11 12	(typ)	$be_0^{C-1}, disp_0^{C-1} + D[c-1] + (B[c-1])$	$\mathbf{I} = 1) \cdot ex_{C-1}, \dots,$		
12	(typ)	$pe_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1] + D[c-1]$	$- (B[c-1]-1) \cdot ex_{C-1}) \}.$		
14 15 16 17		TO MPI_TYPE_CREATE_STRU	IDEXED(count, B, D, oldtype, newtype) is equivalent CT(count, B, D, T, newtype), where each entry of T		
18	4.1.3 Si	ubarray Datatype Constructor			
19 20					
21 22 23	MPI_TYF	PE_CREATE_SUBARRAY(ndim order, oldtype, newtype)	s, array_of_sizes, array_of_subsizes, array_of_starts,		
24 25	IN	ndims	number of array dimensions (positive integer)		
26 27	IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)		
28 29	IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)		
30 31	IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)		
32 33	IN	order	array storage order flag (state)		
34	IN	oldtype	array element datatype (handle)		
35 36	OUT	newtype	new datatype (handle)		
37 38 39 40	<pre>int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>				
41	MPI_Type	c c	rray_of_sizes, array_of_subsizes,		
42 43	INTE	GER, INTENT(IN) :: ndims	<pre>der, oldtype, newtype, ierror) , array_of_sizes(ndims),</pre>		
44	array_of_subsizes(ndims), array_of_starts(ndims), order				
45 46		C(MPI_Datatype), INTENT(IN C(MPI_Datatype), INTENT(OU	V -		
40		GER, OPTIONAL, INTENT(OUT			
48					

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

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

The ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.

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

The array_of_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).

Advice to users. In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is n, then the entry in array_of_starts for that dimension is n-1. (*End of advice to users.*)

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

MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)

MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)

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

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

Let the typemap of oldtype have the form:

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

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 4.4 defines the recursion step when order = MPI_ORDER_C. These equations use the conceptual datatypes lb_marker and ub_marker, see Section 4.1.6 for details.

Unofficial Draft for Comment Only

 $\overline{7}$

1		
2	$\operatorname{Cab}_{\operatorname{conv}}(1 \ (\operatorname{civ}_{\operatorname{civ}}) \ (\operatorname{cub}_{\operatorname{civ}}) \ (\operatorname{cub}_{\operatorname{civ}}) \ (45)$))
3 4	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, $ (4.2)	:)
5	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$	
6	$= \{(lb_marker, 0),$	
7	$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$	
8	$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$	
9	$disp_{n-1} + (start_0 + 1) \times ex), \dots$	
10 11	$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots,$	
11	$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$	
13	$(ub_marker, size_0 \times ex) \}$	
14		
15	Subarray($ndims$, { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.3)	3)
16	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$,
17	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$	
18 19	$= \text{Subarray}(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},$	
20	$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$	
21	$\{start_1, start_2, \dots, start_{ndims-1}\},\$	
22		
23	$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$	
24		• •
25 26	Subarray($ndims$, { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.4)	F)
20	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	
28	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	
29	$= \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$	
30	$\{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\},\$	
31	$\{start_0, start_1, \dots, start_{ndims-2}\},\$	
32 33	$\textbf{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	
34	For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Sec	- .
35	tion 13.9.2.	-
36		
37 38	4.1.4 Distributed Array Datatype Constructor	
39	The distributed array type constructor supports HPF-like [42] data distributions. However,	,
40	unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays	5.
41	Advice to users. One can create an HPF-like file view using this type constructor a	19
42	follows. Complementary filetypes are created by having every process of a group ca	
43 44	this constructor with identical arguments (with the exception of rank which should b	
45	set appropriately). These filetypes (along with identical disp and etype) are then use	
46	to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.	
47	and Section 13.3. Using this view, a collective data access operation (with identical	ıl
48	offsets) will yield an HPF-like distribution pattern. (End of advice to users.)	

Unofficial Draft for Comment Only

MPI_TYP		rank, ndims, array_of_gsizes, array_of_distribs, y_of_psizes, order, oldtype, newtype)	1 2
IN	size	size of process group (positive integer)	3
IN	rank	rank in process group (non-negative integer)	4 5
IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)	6 7
IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)	8 9 10
IN	array_of_distribs	distribution of array in each dimension (array of state)	10
IN	array_of_dargs	distribution argument in each dimension (array of pos- itive integers)	12 13
IN	array_of_psizes	size of process grid in each dimension (array of positive integers)	14 15
IN	order	array storage order flag (state)	16 17
IN	oldtype	old datatype (handle)	18
OUT	newtype	new datatype (handle)	19 20
MPI_Type INTE TYPE	<pre>const int array_c const int array_c int order, MPI_Da e_create_darray(size, r array_of_distribs oldtype, newtype, GER, INTENT(IN) :: si</pre>	ze, rank, ndims, array_of_gsizes(ndims), s(ndims), array_of_dargs(ndims), ndims), order (IN) :: oldtype	21 22 23 24 25 26 27 28 29 30 31 32 33
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	34
INTE	ARRAY_OF_DISTRIBS OLDTYPE, NEWTYPE, CGER SIZE, RANK, NDIMS, ARRAY_OF_DARGS(* NEWTYPE, IERROR	ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,	35 36 37 38 39 40 41
MPI_	TYPE_CREATE_DARRAY	can be used to generate the datatypes corresponding	42

MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See Example 4.7.) 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 in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

1 Advice to users. For both Fortran and C arrays, the ordering of processes in the $\mathbf{2}$ process grid is assumed to be row-major. This is consistent with the ordering used in 3 virtual Cartesian process topologies in MPI. To create such virtual process topologies, 4 or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7. (End of advice to users.) 56 Each dimension of the array can be distributed in one of three ways: 7 8 • MPI_DISTRIBUTE_BLOCK - Block distribution 9 10 • MPI_DISTRIBUTE_CYCLIC - Cyclic distribution 11 • MPI_DISTRIBUTE_NONE - Dimension not distributed. 1213 The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. 14The distribution argument for a dimension that is not distributed is ignored. For any 15dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify 16 $array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].$ 17For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to 18 MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-19RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of 20MPI_DISTRIBUTE_DFLT_DARG. 21The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the stor-22age order. Therefore, arrays described by this type constructor may be stored in Fortran 23(column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN 24 and MPI_ORDER_C. 25This routine creates a new MPI datatype with a typemap defined in terms of a function 26called "cyclic()" (see below). 27Without loss of generality, it suffices to define the typemap for the 28MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used. 29 MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the 30 MPI_DISTRIBUTE_CYCLIC case for dimension i as follows. 31 MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG 32 is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to 33 34 $(array_of_gsizes[i] + array_of_psizes[i] - 1)/array_of_psizes[i].$ 35 If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and 36 MPI_DISTRIBUTE_CYCLIC are equivalent. 37 MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with 38 array_of_dargs[i] set to array_of_gsizes[i]. 39 Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to 40 MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with 41 array_of_dargs[i] set to 1. 42For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined 43 by the following code fragment: 44 45oldtypes[0] = oldtype; 46 for (i = 0; i < ndims; i++) {</pre> 47 oldtypes[i+1] = cyclic(array_of_dargs[i], 48

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

Unofficial Draft for Comment Only

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

Unofficial Draft for Comment Only

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

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

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

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

Address and Size Functions 4.1.5

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the **buf** argument is passed the value MPI_BOTTOM. Note that in Fortran MPI_BOTTOM is not usable for initialization or assignment, see Section 2.5.4.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS. The relative displacement between two absolute addresses can be calculated with the function MPI_AINT_DIFF. A new absolute address as sum of an absolute base address and a relative displacement can be calculated with the function MPI_AINT_ADD. To ensure portability, arithmetic on absolute addresses should not be performed with the intrinsic operators "-" and "+". See also Sections 2.5.6 and 4.1.12 on pages 18 and 117.

44Rationale. Address sized integer values, i.e., MPI_Aint or INTEGER(KIND=MPI_ADDRESS_KIND) values, are signed integers, while absolute addresses are unsigned quantities. Direct arithmetic on addresses stored in address sized signed variables can cause overflows, resulting in undefined behavior. (End of rationale.)

Unofficial Draft for Comment Only

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

2627

28 29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

4546

47

```
1
     MPI_GET_ADDRESS(location, address)
2
       IN
                 location
                                             location in caller memory (choice)
3
       OUT
                 address
                                             address of location (integer)
4
5
6
     int MPI_Get_address(const void *location, MPI_Aint *address)
7
     MPI_Get_address(location, address, ierror)
8
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
9
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
13
          <type> LOCATION(*)
14
          INTEGER IERROR
15
          INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
16
         Returns the (byte) address of location.
17
18
                        In the mpi_f08 module, the location argument is not defined with
           Rationale.
19
           INTENT(IN) because existing applications may use MPI_GET_ADDRESS as a substi-
20
           tute for MPI_F_SYNC_REG that was not defined before MPI-3.0. (End of rationale.)
21
22
23
     Example 4.8 Using MPI_GET_ADDRESS for an array.
^{24}
25
        REAL A(100,100)
26
         INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
27
        CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
        CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
28
29
        DIFF = MPI_AINT_DIFF(I2, I1)
30
     ! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
31
     ! implementation dependent.
32
                              C users may be tempted to avoid the usage of
           Advice to users.
33
34
           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
35
           that the value of a pointer (or the pointer cast to int) be the absolute address of the
36
           object pointed at — although this is commonly the case. Furthermore, referencing
37
           may not have a unique definition on machines with a segmented address space. The
38
           use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such
39
           machines as well. (End of advice to users.)
40
41
           Advice to users.
                              To prevent problems with the argument copying and register
42
           optimization done by Fortran compilers, please note the hints in Sections 18.1.10-
43
           18.1.20. (End of advice to users.)
44
45
          To ensure portability, arithmetic on MPI addresses must be performed using the
46
     MPI_AINT_ADD and MPI_AINT_DIFF functions.
47
48
```

ΜΡΙ ΔΙΙ	NT_ADD(base, d	icn)	1	
			2	
IN	base	base address (integer)	3	
IN	disp	displacement (integer)	4	
			5	
MPI_Air	nt MPI_Aint_ado	d(MPI_Aint base, MPI_Aint disp)	6	
INTEGEF	R(KIND=MPI_ADDA	RESS_KIND) MPI_Aint_add(base, disp)	7	
		_ADDRESS_KIND), INTENT(IN) :: base, disp	8 9	
TNTEGEE	R(KIND=MPT ADDI	RESS_KIND) MPI_AINT_ADD(BASE, DISP)	10	
	· _	_ADDRESS_KIND) BASE, DISP	11	
MP		oduces a new MPI_Aint value that is equivalent to the sum of	12	
	*	nents, where base represents a base address returned by a call to	13	
		d disp represents a signed integer displacement. The resulting ad-	14 15	
dress is valid only at the process that generated base, and it must correspond to a location				
in the same object referenced by base, as described in Section 4.1.12. The addition is per-				
formed i	in a manner that	results in the correct MPI_Aint representation of the output address,	17 18	
as if the	process that ori	ginally produced base had called:	19	
MPT Cot	address((chai	r *) base + disp, &result);	20	
III 1_000		, babe alop, wiebuie,	21	
			22	
ΜΡΙ ΔΙΙ	NT_DIFF(addr1,	(Crbbc	23	
			24	
IN	addr1	minuend address (integer)	25	
IN	addr2	subtrahend address (integer)	26 27	
			28	
MPI_Air	nt MPI_Aint_dia	ff(MPI_Aint addr1, MPI_Aint addr2)	29	
INTEGEF	R(KIND=MPI_ADDE	RESS_KIND) MPI_Aint_diff(addr1, addr2)	30	
INT	TEGER(KIND=MPI	_ADDRESS_KIND), INTENT(IN) :: addr1, addr2	31	
TNTEGEE	KIND=MPT ADDI	RESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)	32	
		_ADDRESS_KIND) ADDR1, ADDR2	33	
			34	
	-	oduces a new MPI_Aint value that is equivalent to the difference	35 36	
		2 arguments, where addr1 and addr2 represent addresses returned DDRESS. The resulting address is valid only at the process that	37	
ē		r2, and addr1 and addr2 must correspond to locations in the same	38	
0		ess, as described in Section $4.1.12$. The difference is calculated in	39	
-	_	the signed difference from addr1 to addr2, as if the process that	40	
		addresses had called (char *) addr1 - (char *) addr2 on the	41	
		to MPI_GET_ADDRESS.	42	
The	e following auxili	ary functions provide useful information on derived datatypes.	43	
			44	
			45 46	
			40 47	

```
1
     MPI_TYPE_SIZE(datatype, size)
2
       IN
                 datatype
                                            datatype (handle)
3
       OUT
                 size
                                            datatype size (integer)
4
5
6
     int MPI_Type_size(MPI_Datatype datatype, int *size)
\overline{7}
     MPI_Type_size(datatype, size, ierror)
8
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
9
          INTEGER, INTENT(OUT) :: size
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
11
12
     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
          INTEGER DATATYPE, SIZE, IERROR
13
14
15
16
     MPI_TYPE_SIZE_X(datatype, size)
17
       IN
                 datatype
                                            datatype (handle)
18
       OUT
19
                size
                                            datatype size (integer)
20
21
     int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
22
     MPI_Type_size_x(datatype, size, ierror)
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
^{24}
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::
                                                             size
25
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
26
27
     MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
28
          INTEGER DATATYPE, IERROR
29
          INTEGER(KIND=MPI_COUNT_KIND) SIZE
30
^{31}
```

MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity. For both functions, if the OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

37 38

4.1.6 Lower-Bound and Upper-Bound Markers

39 It is often convenient to define explicitly the lower bound and upper bound of a type map, 40and override the definition given on page 107. This allows one to define a datatype that has 41 "holes" at its beginning or its end, or a datatype with entries that extend above the upper 42bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. 43 Also, the user may want to overide the alignment rules that are used to compute upper 44bounds and extents. E.g., a C compiler may allow the user to overide default alignment 45rules for some of the structures within a program. The user has to specify explicitly the 46 bounds of the datatypes that match these structures. 47

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

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

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

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

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

Then

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

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least non-negative increment needed to round extent(Typemap) to the next multiple of $\max_i k_i$. In Fortran, 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.)

Advice to implementors. In Fortran, it is generally recommended to use BIND(C) 45 derived types instead of common blocks or SEQUENCE derived types. Therefore it is 46 recommended to calculate the alignments k_i based on BIND(C) derived types. (End 47 of advice to implementors.) 48

Unofficial Draft for Comment Only

 31

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 DOUBLE PRECISION is
! two times the size of a REAL
DOUBLE PRECISION :: p
```

one should define

END TYPE

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

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

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a max_ik_i alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 18.1.15. (End of advice to users.)

4.1.7 Extent and Bounds of Datatypes

```
40
41
42
```

43

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

44	IN	datatype	datatype to get information on (handle)
45 46	OUT	lb	lower bound of datatype (integer)
40	OUT	extent	extent of datatype (integer)
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

Unofficial Draft for Comment Only

4.1. DERIVED DATATYPES

int	nt MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *1b, MPI_Aint *extent)			
MPI_	<pre>PI_Type_get_extent(datatype, lb, extent, ierror)</pre>			
	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent			
	INTEGER, OPTIONAL, INTENT(OU	JT) :: ierror	7	
MPI_	TYPE_GET_EXTENT(DATATYPE, LE	3, EXTENT, IERROR)	8	
	INTEGER DATATYPE, IERROR		9 10	
	INTEGER(KIND=MPI_ADDRESS_KIN	ID) LB, EXTENT	11	
			12	
MPI.	_TYPE_GET_EXTENT_X(datatyp	e, lb, extent)	13	
IN	datatype	datatype to get information on (handle)	14 15	
οι		lower bound of datatype (integer)	16	
OL		extent of datatype (integer)	17	
00		extent of datatype (integer)	18	
int	MPI Type get extent x(MPI Da	atatype datatype, MPI_Count *1b,	19	
	MPI_Count *extent)		20 21	
мрт	Type_get_extent_x(datatype,	1b extent ierror)	21	
· · · · · _	TYPE(MPI_Datatype), INTENT(I		23	
	INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: 1b, extent			
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
MPI_	TYPE_GET_EXTENT_X(DATATYPE,	LB, EXTENT, IERROR)	26 27	
_	INTEGER DATATYPE, IERROR			
	INTEGER(KIND=MPI_COUNT_KIND)	LB, EXTENT	28 29	
	Returns the lower bound and the	extent of datatype (as defined in Equation 4.1).	30	
		parameter cannot express the value to be returned	31	
(e.g.	_	hold the output value), it is set to MPI_UNDEFINED.	32	
,		nt of a datatype, using lower bound and upper bound	33 34	
	*	he stride of successive datatypes that are replicated	35	
by u	atatype constructors, or are replic	eated by the count argument in a send or receive call.	36	
			37	
MPI.	_TYPE_CREATE_RESIZED(oldtyp	e, lb, extent, newtype)	38	
IN	oldtype	input datatype (handle)	39 40	
IN	lb	new lower bound of datatype (integer)	40	
IN	extent	new extent of datatype (integer)	42	
οι	T newtype	output datatype (handle)	43	
		- · · · /	44 45	
int	int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,			
	MPI_Aint extent, M	PI_Datatype *newtype)	46 47	
MPI_	MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) 48			

Unofficial Draft for Comment Only

```
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                              lb, extent
\mathbf{2}
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
3
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
          INTEGER, OPTIONAL, INTENT(OUT) ::
4
                                                 ierror
5
     MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
6
          INTEGER OLDTYPE, NEWTYPE, IERROR
7
          INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
8
9
         Returns in newtype a handle to a new datatype that is identical to oldtype, except that
10
     the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb
11
     + extent. Any previous lb and ub markers are erased, and a new pair of lower bound and
12
     upper bound markers are put in the positions indicated by the lb and extent arguments.
13
     This affects the behavior of the datatype when used in communication operations, with
14
     count > 1, and when used in the construction of new derived datatypes.
15
16
     4.1.8
            True Extent of Datatypes
17
     Suppose we implement gather (see also Section 5.5) as a spanning tree implemented on
18
     top of point-to-point routines. Since the receive buffer is only valid on the root pro-
19
     cess, one will need to allocate some temporary space for receiving data on intermedi-
20
     ate nodes. However, the datatype extent cannot be used as an estimate of the amount
21
     of space that needs to be allocated, if the user has modified the extent, for example
22
     by using MPI_TYPE_CREATE_RESIZED. The functions MPI_TYPE_GET_TRUE_EXTENT
23
     and MPI_TYPE_GET_TRUE_EXTENT_X are provided which return the true extent of the
24
     datatype.
25
26
27
     MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
28
       IN
                datatype
                                            datatype to get information on (handle)
29
30
       OUT
                true_lb
                                            true lower bound of datatype (integer)
^{31}
       OUT
                true_extent
                                            true size of datatype (integer)
32
33
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
34
                    MPI_Aint *true_extent)
35
36
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
37
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
39
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
40
     MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
41
          INTEGER DATATYPE, IERROR
42
          INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
43
44
45
46
47
48
```

MPI_TYF	PE_GET_TRUE_EXTE	NT_X(datatype, true_lb, true_extent)
IN	datatype	datatype to get information on (handle)
OUT	true_lb	true lower bound of datatype (integer)
OUT	true_extent	true size of datatype (integer)
	-	
nt MPI_	Type_get_true_exten	nt_x(MPI_Datatype datatype, MPI_Count *true_lb,
	MPI_Count *tr	
PT Type	get true extent x	(datatype, true_lb, true_extent, ierror)
	-	<pre>TENT(IN) :: datatype</pre>
	• 1	_KIND), INTENT(OUT) :: true_lb, true_extent
INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror
PI_TYPE	GET_TRUE_EXTENT X	(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
	CGER DATATYPE, IERRO	
INTE	GER(KIND=MPI_COUNT	_KIND) TRUE_LB, TRUE_EXTENT
true	lb returns the offset of	the lowest unit of store which is addressed by the datatype,
		responding typemap, ignoring explicit lower bound mark-
		ie size of the datatype, i.e., the extent of the correspond-
0 0 *	*	ower bound and upper bound markers, and performing no
ounding	for alignment. If the t	ypemap associated with datatype is
Typ	$pemap = \{(type_0, disp_0)\}$),, $(type_{n-1}, disp_{n-1})$ }
hen		
true	$e_{lb}(Typemap) = min_{s}$	$_{j}\{disp_{j} \; : \; type_{j} \neq lb_marker, ub_marker\},$
true	$e_ub(Typemap) = max$	$x_i \{ disp_i + sizeof(type_i) : type_i \neq lb_marker, ub_marker \},$
	- ()1 () () ()	
nd		
true	e extent(Tuneman) =	$true_ub(Typemap) - true_lb(typemap).$
	· · · · · · · · · · · · · · · · · · ·	
	=	with the definitions in Section $4.1.6$ and Section $4.1.7$, which
	the function MPI_TYP	E_GEI_EXIENI.) nimum number of bytes of memory necessary to hold a
	uncompressed.	minum number of bytes of memory necessary to hold a
υ ,	•	er OUT parameter cannot express the value to be returned
		all to hold the output value), it is set to MPI_UNDEFINED.
.1.9 C	ommit and Free	
dataty	pe object has to be c o	ommitted before it can be used in a communication. As
		uctors, uncommitted and also committed datatypes can be
-		nit basic datatypes. They are "pre-committed."

 48

```
1
     MPI_TYPE_COMMIT(datatype)
\mathbf{2}
       INOUT
                datatype
                                            datatype that is committed (handle)
3
4
     int MPI_Type_commit(MPI_Datatype *datatype)
5
6
     MPI_Type_commit(datatype, ierror)
7
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_TYPE_COMMIT(DATATYPE, IERROR)
10
         INTEGER DATATYPE, IERROR
11
12
         The commit operation commits the datatype, that is, the formal description of a com-
13
     munication buffer, not the content of that buffer. Thus, after a datatype has been commit-
14
     ted, it can be repeatedly reused to communicate the changing content of a buffer or, indeed,
15
     the content of different buffers, with different starting addresses.
16
17
                                     The system may "compile" at commit time an internal
          Advice to implementors.
18
          representation for the datatype that facilitates communication, e.g., change from a
19
          compacted representation to a flat representation of the datatype, and select the most
20
          convenient transfer mechanism. (End of advice to implementors.)
21
         MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent
22
23
     to a no-op.
24
     Example 4.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.
25
26
     INTEGER type1, type2
27
     CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
28
                     ! new type object created
29
     CALL MPI_TYPE_COMMIT(type1, ierr)
30
                     ! now type1 can be used for communication
^{31}
     type2 = type1
32
                     ! type2 can be used for communication
33
                     ! (it is a handle to same object as type1)
34
     CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
35
                     ! new uncommitted type object created
36
     CALL MPI_TYPE_COMMIT(type1, ierr)
37
                     ! now type1 can be used anew for communication
38
39
40
     MPI_TYPE_FREE(datatype)
41
42
       INOUT
                datatype
                                             datatype that is freed (handle)
43
44
     int MPI_Type_free(MPI_Datatype *datatype)
45
     MPI_Type_free(datatype, ierror)
46
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_TYPE_FREE(DATATYPE, IERROR) INTEGER DATATYPE, IERROR

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one 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 $oldtype\xspace$ (handle)

```
int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_Type_dup(oldtype, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
```

MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR

INTEGER, OPTIONAL, INTENT(OUT) ::

MPI_TYPE_DUP is a type constructor which duplicates the existing oldtype with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as oldtype and any copied cached information, see Section 6.7.4. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The newtype has the same committed state as the old oldtype.

ierror

4.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype 44 argument is required. A call of the form MPI_SEND(buf, count, datatype, ...), where count > 45 1, is interpreted as if the call was passed a new datatype which is the concatenation of count 46 copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to, 47

1

 $\mathbf{2}$

3

4

5

6

7

8 9

10

11

12

13

14

15 16

17 18 19

25 26

27

28

29

30

31

32 33

34

35

36

37

38

39

40

41 42

43

12	MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
3	MPI_TYPE_COMMIT(newtype)
4	MPI_SEND(buf, 1, newtype, dest, tag, comm) MPI_TYPE_FREE(newtype).
5	
6	Similar statements apply to all other communication functions that have a count and
7	datatype argument.
8	Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is
9	executed, where datatype has type map,
10	$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$
11	and extent <i>extent</i> . (Explicit lower bound and upper bound markers are not listed in the
12	type map, but they affect the value of <i>extent</i> .) The send operation sends $n \cdot \text{count}$ entries,
13	where entry $i \cdot n + j$ is at location $addr_{i,j} = buf + extent \cdot i + disp_j$ and has type $type_j$, for
14	$i = 0, \dots, \text{count} - 1$ and $j = 0, \dots, n-1$. These entries need not be contiguous, nor distinct;
15 16	their order can be arbitrary.
17	The variable stored at address $addr_{i,j}$ in the calling program should be of a type that
18	matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains
19	$n \cdot \text{count entries}$, where entry $i \cdot n + j$ has type $type_j$.
20	Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag,
21	comm, status) is executed, where datatype has type map,
22	$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$
23	with extent <i>extent</i> . (Again, explicit lower bound and upper bound markers are not listed in
24	the type map, but they affect the value of <i>extent</i> .) This receive operation receives $n \cdot \text{count}$
25	entries, where entry $i \cdot n + j$ is at location buf $+ extent \cdot i + disp_j$ and has type $type_j$. If the
26	incoming message consists of k elements, then we must have $k \leq n \cdot \text{count}$; the $i \cdot n + j$ -th
27	element of the message should have a type that matches $type_j$.
28	Type matching is defined according to the type signature of the corresponding
29	datatypes, that is, the sequence of basic type components. Type matching does not depend
30	on some aspects of the datatype definition, such as the displacements (layout in memory)
31 32	or the intermediate types used.
33	Example 4.11 This example shows that type matching is defined in terms of the basic
34	types that a derived type consists of.
35	
36	CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2,)
37	CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4,)
38	CALL MPI_TYPE_CONTIGUOUS(2, type2, type22,)
39	
40	CALL MPI_SEND(a, 4, MPI_REAL,)
41	CALL MPI_SEND(a, 2, type2,)
42	CALL MPI_SEND(a, 1, type22,)
43	CALL MPI_SEND(a, 1, type4,)
44	••••
45	CALL MPI_RECV(a, 4, MPI_REAL,)
46 47	CALL MPI_RECV(a, 2, type2,)
47 48	CALL MPI_RECV(a, 1, type22,)
10	CALL MPI_RECV(a, 1, type4,)

Each of the sends matches any of the receives.

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

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

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

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where $0 \le k \le \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.

MPI_GET_ELEMENTS(status, datatype, count)

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

MPI_Get_elements(status, datatype, count, ierror)
TYPE(MPI_Status), INTENT(IN) :: status
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(OUT) :: count
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_GET_ELEMENTS_X(status, datatype, count)

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

MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)

Unofficial Draft for Comment Only

```
1
          INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
\mathbf{2}
          INTEGER(KIND=MPI_COUNT_KIND) COUNT
3
         The datatype argument should match the argument provided by the receive call that
4
     set the status variable. For both functions, if the OUT parameter cannot express the value
5
     to be returned (e.g., if the parameter is too small to hold the output value), it is set to
6
     MPI_UNDEFINED.
7
          The previously defined function MPI_GET_COUNT (Section 3.2.5), has a different be-
8
     havior. It returns the number of "top-level entries" received, i.e. the number of "copies" of
9
     type datatype. In the previous example, MPI_GET_COUNT may return any integer value
10
     k, where 0 \le k \le \text{count}. If MPI_GET_COUNT returns k, then the number of basic elements
11
     received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is
12
     n \cdot k. If the number of basic elements received is not a multiple of n, that is, if the receive
13
     operation has not received an integral number of datatype "copies," then MPI_GET_COUNT
14
     sets the value of count to MPI_UNDEFINED.
15
16
     Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.
17
18
     . . .
     CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
19
     CALL MPI_TYPE_COMMIT(Type2, ierr)
20
     . . .
21
     CALL MPI_COMM_RANK(comm, rank, ierr)
22
     IF (rank.EQ.0) THEN
23
            CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
24
            CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
25
26
     ELSE IF (rank.EQ.1) THEN
            CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
27
            CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                               ! returns i=1
28
            CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                               ! returns i=2
29
            CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
30
            CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                               ! returns i=MPI_UNDEFINED
31
            CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                              ! returns i=3
32
     END IF
33
34
         The functions MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X can also be used
35
     after a probe to find the number of elements in the probed message. Note that the
36
     MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return the same
37
     values when they are used with basic datatypes as long as the limits of their respective
38
     count arguments are not exceeded.
39
40
           Rationale. The extension given to the definition of MPI_GET_COUNT seems natural:
41
           one would expect this function to return the value of the count argument, when the
42
           receive buffer is filled. Sometimes datatype represents a basic unit of data one wants
43
           to transfer, for example, a record in an array of records (structures). One should be
44
           able to find out how many components were received without bothering to divide by
45
           the number of elements in each component. However, on other occasions, datatype
46
           is used to define a complex layout of data in the receiver memory, and does not
47
           represent a basic unit of data for transfers. In such cases, one needs to use the
48
           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.

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

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

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

Advice to implementors. There is no need to distinguish (absolute) addresses and ⁴⁶ (relative) displacements on a machine with contiguous address space: MPI_BOTTOM ⁴⁷ is zero, and both addresses and displacements are integers. On machines where the ⁴⁸

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 24

 31

distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (End of advice to implementors.)

4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There 6 are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Further-8 more, a number of tools wish to display internal information about a datatype. To achieve 9 this, datatype decoding functions are provided. The two functions in this section are used 10 together to decode datatypes to recreate the calling sequence used in their initial defini-11 tion. These can be used to allow a user to determine the type map and type signature of a 12datatype. 13

1415

16

1

 $\mathbf{2}$

3 4

5

7

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

17		combiner)	
18	IN	datatype	datatype to access (handle)
19	OUT	num_integers	number of input integers used in the call constructing
20			combiner (non-negative integer)
21 22	OUT	num_addresses	number of input addresses used in the call construct-
22			ing combiner (non-negative integer)
24	OUT	num_datatypes	number of input datatypes used in the call construct-
25			ing combiner (non-negative integer)
26	OUT	combiner	combiner (state)
27			
28 29	int MPI_T	<pre>ype_get_envelope(MPI_Data</pre>	type datatype, int *num_integers,
29 30		<pre>int *num_addresses, :</pre>	<pre>int *num_datatypes, int *combiner)</pre>
31			
32		combiner, ierror)	
33	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
34	INTEG		ntegers, num_addresses, num_datatypes,
35		combiner	
36	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror

37 MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, 38 COMBINER, IERROR) 39

INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the num-42ber and type of input arguments used in the call that created the datatype. The number-of-43 arguments values returned can be used to provide sufficiently large arrays in the decoding 44routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is 45described below. The combiner reflects the MPI datatype constructor call that was used in 4647creating datatype.

48

40

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.

MPI_COMBINER_NAMED	a named predefined datatype	14
MPI_COMBINER_DUP	MPI_TYPE_DUP	15
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	16
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	17
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	18
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	19
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	20
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	21
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	22
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	23
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	24
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	25
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	26
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX	27
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	28
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	29

Table 4.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

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. $\overline{7}$

```
1
     MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes,
\mathbf{2}
                     array_of_integers, array_of_addresses, array_of_datatypes)
3
       IN
                 datatype
                                             datatype to access (handle)
4
                 max_integers
       IN
                                             number of elements in array_of_integers (non-negative
5
                                             integer)
6
7
       IN
                 max_addresses
                                             number of elements in array_of_addresses (non-negative
8
                                             integer)
9
       IN
                 max_datatypes
                                             number of elements in array_of_datatypes (non-negative
10
                                             integer)
11
       OUT
                 array_of_integers
                                             contains integer arguments used in constructing
12
                                             datatype (array of integers)
13
14
       OUT
                 array_of_addresses
                                             contains address arguments used in constructing
15
                                             datatype (array of integers)
16
       OUT
                 array_of_datatypes
                                             contains datatype arguments used in constructing
17
                                             datatype (array of handles)
18
19
     int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
20
                     int max_addresses, int max_datatypes, int array_of_integers[],
21
                    MPI_Aint array_of_addresses[],
22
                    MPI_Datatype array_of_datatypes[])
23
24
     MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
25
                    array_of_integers, array_of_addresses, array_of_datatypes,
26
                     ierror)
27
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
          INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
29
          INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
30
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
31
                     array_of_addresses(max_addresses)
32
          TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
35
                     ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
36
                     IERROR)
37
          INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
38
                     ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
39
          INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
40
41
          datatype must be a predefined unnamed or a derived datatype; the call is erroneous if
42
     datatype is a predefined named datatype.
43
          The values given for max_integers, max_addresses, and max_datatypes must be at least as
44
     large as the value returned in num_integers, num_addresses, and num_datatypes, respectively,
45
     in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.
46
47
           Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for
48
           error checking in the call. (End of rationale.)
```

CHAPTER 4. DATATYPES

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

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

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

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

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

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

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

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

The following defines what values are placed in each entry of the returned arrays ⁴⁵ depending on the datatype constructor used for datatype. It also specifies the size of the ⁴⁶ arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, ⁴⁷ the following calls were made: ⁴⁸

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	PARAMETER (LARGE = 1000)			
2	INTEGER TYPE, NI, NA, ND, COMBINER, I(I	ARGE) D(LARGE) TERROR		
3	INTEGER (KIND=MPI_ADDRESS_KIND) A(LARGE			
		<u>-</u>)		
4	! CONSTRUCT DATATYPE TYPE (NOT SHOWN)			
5	CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)			
6	IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE	E) .OR. (ND .GT. LARGE)) THEN		
7	WRITE (*, *) "NI, NA, OR ND = ", NI,			
8	" RETURNED BY MPI_TYPE_GET_ENVELOPE 1			
		-		
9	CALL MPI_ABORT(MPI_COMM_WORLD, 99, IE	CRRUR)		
10	ENDIF			
11	CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA	A, ND, I, A, D, IERROR)		
12				
13	or in C the analogous calls of:			
14	-			
	#define LARGE 1000			
15	<pre>int ni, na, nd, combiner, i[LARGE];</pre>			
16	MPI_Aint a[LARGE];			
17	-			
18	<pre>MPI_Datatype type, d[LARGE];</pre>			
19	<pre>/* construct datatype type (not shown) */</pre>			
	MPI_Type_get_envelope(type, ∋, &na, &nd, &c	combiner);		
20	if ((ni > LARGE) (na > LARGE) (nd > LAF	RGE)) {		
21	<pre>fprintf(stderr, "ni, na, or nd = %d %d %d</pre>			
22	fprintf(stderr, "MPI_Type_get_envelope is	·		
23		s larger than LANGE - //d li ,		
24	LARGE);			
	MPI_Abort(MPI_COMM_WORLD, 99);			
25	};			
26	MPI_Type_get_contents(type, ni, na, nd, i, a, d);			
27				
28	In the descriptions that follow, the lower case nat	me of arguments is used.		
29	If combiner is MPI_COMBINER_NAMED then it	is erroneous to call		
30	MPI_TYPE_GET_CONTENTS.			
31	If combiner is MPI_COMBINER_DUP then			
	II COMDINEL IS MFI_COMDINEL_DOF CHEM			
32	Constructor argument C F	Fortran location		
33				
34	oldtype d[0]	D(1)		
35	and ni 0 na 0 nd 1			
36	and $ni = 0$, $na = 0$, $nd = 1$.			
	If combiner is MPI_COMBINER_CONTIGUOUS then	n		
37				
38		Fortran location		
39	count i[0]	I(1)		
40	oldtype d[0]	$\mathrm{D}(1)$		
41				
42	and $ni = 1$, $na = 0$, $nd = 1$.			
	If combiner is MPI_COMBINER_VECTOR then			
43				
44	Constructor argument C Fortran location			
45	count i[0]	I(1)		
46				
47	blocklength i[1]	I(2)		
	stride i[2]	I(3)		
48	oldtype $d[0]$	D(1)		

and ni = 3, na = 0, nd = 1.

If combiner is $\mathsf{MPI_COMBINER_HVECTOR}$ then

Constructor argument	С	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	a[0]	A(1)
oldtype	d[0]	D(1)

and ni = 2, na = 1, nd = 1.

If combiner is MPI_COMBINER_INDEXED then

Constructor argument	С	Fortran location
count	i[0]	I(1)
$array_of_blocklengths$	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
$array_{of_displacements}$	i[i[0]+1] to $i[2*i[0]]$	I(I(1)+2) to $I(2*I(1)+1)$
oldtype	d[0]	D(1)

and $ni = 2^{*}count+1$, na = 0, nd = 1.

_

If combiner is MPI_COMBINER_HINDEXED then

Constructor argument	С	Fortran location
count	i[0]	I(1)
$array_of_blocklengths$	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
array_of_displacements	a[0] to $a[i[0]-1]$	A(1) to $A(I(1))$
oldtype	d[0]	D(1)

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

Constructor argument	С	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
$array_of_displacements$	i[2] to $i[i[0]+1]$	I(3) to $I(I(1)+2)$
oldtype	d[0]	D(1)

and ni = count+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED_BLOCK then

Constructor argument	С	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
$array_of_displacements$	a[0] to $a[i[0]-1]$	A(1) to $A(I(1))$
oldtype	d[0]	D(1)

and ni = 2, na = count, nd = 1.

If combiner is MPI_COMBINER_STRUCT then

	Constructor one	mana anat	C	Fortron loost	ian
	Constructor arg	gument	C	Fortran locat	.1011
	count	ongthe	i[0]	I(1) $I(2) \neq I(I(1))$	+ 1)
	array_of_blockl array_of_displa	-	i[1] to $i[i[0]]$,
	array_of_types		[0] to $a[i[0]-[0]$ to $d[i[0]-$		
	array_or_types				<u>,+//</u>
	nt+1, $na = count$,			
If combine	er is MPI_COMBI	NER_SUBAR	RAY then		
Consti	ructor argument		C	Fortran lo	cation
ndims	_	i	[0]	I(1)	
array_	of_sizes		5 i[i[0]]	I(2) to $I(I($	(1)+1)
array_	of_subsizes		to i[2*i[0]]		· / /
	of_starts		to i[3*i[0]]		
order	_		[0]+1]	I(3*I(1))	, ,
oldtyp	e	-	[0]	D(1)	-
	$\lim_{n \to \infty} +2$, $na = 0, 1$		_		
If combine	er is MPI_COMBI	NER_DARRA	Y then		
Constru	ictor argument		С	Fortran l	ocation
size	0		[0]	I(1	
rank			[1]	-(- I(2	,
ndims			[2]	-(- I(3	,
array_o	f_gsizes		i[i[2]+2]	I(4) to $I($,
	0				× / · /
arrav o	f_distribs	i i 2 +3 to) $1 Z 1 Z + Z $	I(I(3)+4) to	I(2*I(3)+3)
-	f_distribs f_dargs			I(I(3)+4) to I $I(2*I(3)+4)$ to	
array_o	f_dargs	i[2*i[2]+3]	to $i[3*i[2]+2$] $I(2*I(3)+4)$ to	I(3*I(3)+3)
array_of array_of	f_dargs	i[2*i[2]+3] = i[3*i[2]+3] = i[3*i[3] = i[3*i[2]+3] = i[3*i[3]+3] = i[3*i[3]+3] = i[3*i[3]+3] = i[3	to $i[3*i[2]+2$ to $i[4*i[2]+2$	$\begin{bmatrix} I(2*I(3)+4) & to \\ I(3*I(3)+4) & to \\ \end{bmatrix}$	I(3*I(3)+ I(4*I(3)+
array_of array_of order	f_dargs f_psizes	i[2*i[2]+3] i[3*i[2]+3] i[4*i]	to $i[3*i[2]+2$ to $i[4*i[2]+2$ [2]+3]	$ \begin{bmatrix} I(2*I(3)+4) & to \\ I(3*I(3)+4) & to \\ I(4*I(3)+4) & to \\ I(4*I(3)$	I(3*I(3)+ > $I(4*I(3)+$ B)+4)
array_o array_o order oldtype	f_dargs f_psizes	i[2*i[2]+3] i[3*i[2]+3] i[4*i] d	to $i[3*i[2]+2$ to $i[4*i[2]+2$	$\begin{bmatrix} I(2*I(3)+4) & to \\ I(3*I(3)+4) & to \\ \end{bmatrix}$	I(3*I(3)+ I(4*I(3)+ B)+4)
array_of array_of order oldtype and ni = 4*nd	f_dargs f_psizes lims+4, na = 0, r	$i[2^{*}i[2]+3] + i[3^{*}i[2]+3] + i[4^{*}i] + i[4^{*$	to $i[3^*i[2]+2$ to $i[4^*i[2]+2$ [2]+3] [0]	$ \begin{bmatrix} I(2*I(3)+4) & to \\ I(3*I(3)+4) & to \\ I(4*I(3)+4) & to \\ I(4*I(3)$	I(3*I(3)+ > $I(4*I(3)+$ B)+4)
array_of array_of order oldtype and ni = 4*nd	f_dargs f_psizes	$i[2^{*}i[2]+3] + i[3^{*}i[2]+3] + i[4^{*}i] + i[4^{*$	to $i[3^*i[2]+2$ to $i[4^*i[2]+2$ [2]+3] [0]	$ \begin{bmatrix} I(2*I(3)+4) & to \\ I(3*I(3)+4) & to \\ I(4*I(3)+4) & to \\ I(4*I(3)$	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI Construct	$i[2^{*}i[2]+3] + i[3^{*}i[2]+3] + i[4^{*}i] + i[4^{*$	$\frac{1}{2} = \frac{1}{2} = \frac{1}$] I(2*I(3)+4) to] I(3*I(3)+4) to I(4*I(3) D(2) prtran location	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI $\frac{Construct}{p}$	i[2*i[2]+3] + i[3*i[2]+3] + i[4*i] +	$\frac{1}{100} \frac{1}{100} \frac{1}$	$\frac{[] I(2*I(3)+4) to}{I(3*I(3)+4) to}$ $I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(3+1) I$	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI Construct	i[2*i[2]+3] + i[3*i[2]+3] + i[4*i] +	$\frac{1}{2} = \frac{1}{2} = \frac{1}$] I(2*I(3)+4) to] I(3*I(3)+4) to I(4*I(3) D(2) prtran location	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_o array_o order oldtype and ni = 4*nd If combine	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI $\frac{Construc}{p}$ r	i[2*i[2]+3] + i[3*i[2]+3] + i[4*i] +	$\frac{1}{100} \frac{1}{100} \frac{1}$	$\frac{[] I(2*I(3)+4) to}{I(3*I(3)+4) to}$ $I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(4*I(3)+4) I(3) I(3) I(3) I(3) I(3) I(3) I(3) I(3$	I(3*I(3)+ I(4*I(3)+ B)+4)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI $\frac{Construct}{p}$ r a = 0, nd = 0.	i[2*i[2]+3] + i[3*i[2]+3] + i[4*i] + i[4*i] + d nd = 1. NER_F90_RE	$\frac{\text{co i}[3^*i[2]+2}{\text{co i}[4^*i[2]+2}$ $\frac{\text{co i}[4^*i[2]+2}{\text{[2]}+3]}$ $\frac{\text{[0]}}{\text{AL then}}$ $\frac{\text{nt } C \text{Fe}}{\text{i}[0]}$ $\frac{\text{i}[1]}{\text{i}[1]}$	$\frac{\left[\begin{array}{c} I(2^{*}I(3)+4) \text{ to} \\ I(3^{*}I(3)+4) \text{ to} \\ I(4^{*}I(3)+4) t$	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI $\frac{Construc}{p}$ r	i[2*i[2]+3] + i[3*i[2]+3] + i[4*i] + i[4*i] + d nd = 1. NER_F90_RE	$\frac{\text{co i}[3^*i[2]+2}{\text{co i}[4^*i[2]+2}$ $\frac{\text{co i}[4^*i[2]+2}{\text{[2]}+3]}$ $\frac{\text{[0]}}{\text{AL then}}$ $\frac{\text{nt } C \text{Fe}}{\text{i}[0]}$ $\frac{\text{i}[1]}{\text{i}[1]}$	$\frac{\left[\begin{array}{c} I(2^{*}I(3)+4) \text{ to} \\ I(3^{*}I(3)+4) \text{ to} \\ I(4^{*}I(3)+4) t$	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI $\frac{Construc}{p}$ r a = 0, nd = 0. er is MPI_COMBI	i[2*i[2]+3] i[3*i[2]+3] i[4*i] d d d d d d d d	$\frac{1}{100} i[3^*i[2]+2$ $\frac{1}{100} i[4^*i[2]+2$ $\frac{1}{100} i[2]+3]$ $\frac{1}{100} i[1]$ $\frac{1}{100} i[1]$ $\frac{1}{100} i[1]$	$ \begin{array}{c} I(2^{*}I(3)+4) \text{ to} \\ I(3^{*}I(3)+4) \text{ to} \\ I(4^{*}I(3)+4) \text{ to} $	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na	f_dargs f_psizes $dims+4, na = 0, rac{1}{2}$ er is MPI_COMBI $\frac{Construct}{p}$ r a = 0, nd = 0. er is MPI_COMBI Construct	i[2*i[2]+3] + i[3*i[2]+3] + i[4*i] + i[4*i] + d nd = 1. NER_F90_RE	$\frac{1}{100} = \frac{1}{100} = \frac{1}$	$\begin{array}{c} I(2^*I(3)+4) \text{ to}\\ I(3^*I(3)+4) \text{ to}\\ I(4^*I(3)+4) \text{ to}\\ I($	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na	f_dargs f_psizes lims+4, na = 0, r er is MPI_COMBI $\frac{Construc}{p}$ r a = 0, nd = 0. er is MPI_COMBI	i[2*i[2]+3] i[3*i[2]+3] i[4*i] d d d d d d d d	$\frac{1}{100} = \frac{1}{100} = \frac{1}$	$\frac{\left[\begin{array}{c} I(2*I(3)+4) \text{ to} \\ I(3*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(1) \\ I(2) \\ I(1) \\ I(2) \\ I(1) $	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na If combine	f_dargs f_psizes $lims+4, na = 0, rer is MPI_COMBI\frac{Construc}{p}ra = 0, nd = 0.er is MPI_COMBI\frac{Construc}{p}r$	i[2*i[2]+3] i[3*i[2]+3] i[4*i] d d d d d d d d	$\frac{1}{100} = \frac{1}{100} = \frac{1}$	$\begin{array}{c} I(2^*I(3)+4) \text{ to}\\ I(3^*I(3)+4) \text{ to}\\ I(4^*I(3)+4) \text{ to}\\ I($	I(3*I(3)+ I(4*I(3)+ B)+4)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na If combine and ni = 2, na	f_dargs f_psizes $dims+4, na = 0, r$ er is MPI_COMBI $\frac{Construc}{p}$ r a = 0, nd = 0. er is MPI_COMBI $\frac{Construc}{p}$ r a = 0, nd = 0.	i[2*i[2]+3] 1 i[3*i[2]+3] 1 i[4*i d nd = 1. NER_F90_RE ctor argumes	$\frac{1}{10} = \frac{1}{10} $	$\frac{\left[\begin{array}{c} I(2*I(3)+4) \text{ to} \\ I(3*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(1) \\ I(2) \\ I(1) \\ I(2) \\ I(1) $	I(3*I(3)+ I(4*I(3)+ B)+4)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na If combine and ni = 2, na	f_dargs f_psizes $lims+4, na = 0, rer is MPI_COMBI\frac{Construc}{p}ra = 0, nd = 0.er is MPI_COMBI\frac{Construc}{p}r$	i[2*i[2]+3] 1 i[3*i[2]+3] 1 i[4*i d nd = 1. NER_F90_RE ctor argumes	$\frac{1}{10} = \frac{1}{10} $	$\frac{\left[\begin{array}{c} I(2*I(3)+4) \text{ to} \\ I(3*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(1) \\ I(2) \\ I(1) \\ I(2) \\ I(1) $	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na If combine and ni = 2, na	f_dargs f_psizes $dims+4, na = 0, rac{1}{2}$ er is MPI_COMBI $\frac{Construct}{p}$ r a = 0, nd = 0. er is MPI_COMBI $\frac{Construct}{p}$ r a = 0, nd = 0. er is MPI_COMBI	i[2*i[2]+3] t i[3*i[2]+3] t i[4*i d nd = 1. NER_F90_RE etor argumen NER_F90_CC etor argumen	$\frac{1}{10} = \frac{1}{10} $	$\frac{\left[\begin{array}{c} I(2*I(3)+4) \text{ to} \\ I(3*I(3)+4) \text{ to} \\ I(4*I(3)-4) \text{ to} \\ I(4*I(3)-4) \text{ to} \\ I(4*I(3)-4) \text{ to} \\ I(1) \\ I(1) \\ I(2) \end{array}\right]}{\text{prtran location}}$	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)
array_of array_of order oldtype and ni = 4*nd If combine and ni = 2, na If combine and ni = 2, na	f_dargs f_psizes $dims+4, na = 0, rac{1}{2}$ er is MPI_COMBI $\frac{Construct}{p}$ r a = 0, nd = 0. er is MPI_COMBI $\frac{Construct}{p}$ r a = 0, nd = 0. er is MPI_COMBI	i[2*i[2]+3] 1 i[3*i[2]+3] 1 i[4*i d nd = 1. NER_F90_RE ctor argumes	$\frac{1}{10} = \frac{1}{10} $	$\frac{\left[\begin{array}{c} I(2*I(3)+4) \text{ to} \\ I(3*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(4*I(3)+4) \text{ to} \\ I(1) \\ I(2) \\ I(1) \\ I(2) \\ I(1) $	I(3*I(3)+3) I(4*I(3)+3) I(4*I(3)+3)

	ni = 1, na = 0,		.1		1 2
	If combiner is N	IPI_COMBINER_RESIZED	tnen		3
		Constructor argument	С	Fortran location	4
		lb	a[0]	A(1)	5
		extent	a[1]	A(2)	6
		oldtype	d[0]	D(1)	7
					8
and	ni = 0, na = 2,	$\mathrm{nd} = 1.$			9
					10
4.1.1	4 Examples				1
The	following examp	oles illustrate the use of o	lerived	l datatypes.	1:
	0 1				1:
Exa	mple 4.13 Sen	d and receive a section of	of a 3D) array.	1
					10
		100,100), e(9,9,9)		o munople ionn	1
		slice, twoslice, thre ND=MPI_ADDRESS_KIND)		-	1
		tus(MPI_STATUS_SIZE)	10, :	sizeoirear	1
					20
С	extract the	section a(1:17:2, 3	:11. 2	2:10)	2
C		t in e(:,:,:).	· , -		23
					23
	CALL MPI_CO	MM_RANK(MPI_COMM_WORI	LD, my	vrank, ierr)	2
					2
	CALL MPI_TY	PE_GET_EXTENT(MPI_REA	AL, 11	o, sizeofreal, ierr)	2
					2
С		type for a 1D section			2
	CALL MPI_TY	PE_VECTOR(9, 1, 2, MI	PI_RE	AL, oneslice, ierr)	3
a		toma fam a OD acation	_		3
С		type for a 2D section		00*sizeofreal, oneslice,	3:
	CALL MIT_II			e, ierr)	3
			001100	, 1011)	34
С	create data	type for the entire a	sectio	on	3
		• -		00*100*sizeofreal, twoslice,	3
		th	reesli	ice, ierr)	3
					3
	CALL MPI_TY	PE_COMMIT(threeslice	, ierı	c)	3
	CALL MPI_SE			slice, myrank, 0, e, 9*9*9,	4
		MPI_REAL, myra	ank, (), MPI_COMM_WORLD, status, ierr)	4
					4
Exa	mple 4.14 Con	by the (strictly) lower tri	angula	ar part of a matrix.	4
			0	L	4
					4
					4
					4

```
1
           REAL a(100,100), b(100,100)
\mathbf{2}
           INTEGER disp(100), blocklen(100), ltype, myrank, ierr
3
           INTEGER status(MPI_STATUS_SIZE)
4
5
     С
           copy lower triangular part of array a
6
     С
           onto lower triangular part of array b
7
8
           CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
9
10
     С
           compute start and size of each column
11
           DO i=1, 100
12
             disp(i) = 100*(i-1) + i
13
             blocklen(i) = 100-i
14
           END DO
15
16
     С
           create datatype for lower triangular part
17
           CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)
18
19
           CALL MPI_TYPE_COMMIT(ltype, ierr)
20
           CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1,
21
                              ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
22
23
     Example 4.15 Transpose a matrix.
24
           REAL a(100,100), b(100,100)
25
           INTEGER row, xpose, myrank, ierr
26
           INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
27
           INTEGER status(MPI_STATUS_SIZE)
28
29
     С
           transpose matrix a onto b
30
31
           CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
32
33
           CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
34
35
     С
           create datatype for one row
36
           CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
37
38
     С
           create datatype for matrix in row-major order
39
           CALL MPI_TYPE_CREATE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
40
41
           CALL MPI_TYPE_COMMIT(xpose, ierr)
42
43
     С
           send matrix in row-major order and receive in column major order
44
           CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100,
45
                              MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
46
47
48
     Example 4.16 Another approach to the transpose problem:
```

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

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

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

```
1
     MPI_Type_create_hvector(1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
\mathbf{2}
     MPI_Type_commit(&Allpairs);
3
     MPI_Send(particle[0].d, 1, Allpairs, dest, tag, comm);
4
5
           /* an alternative solution to 4.3 */
6
7
     MPI_Datatype Twodouble;
8
9
     MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
10
^{11}
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
12
                                 the extent of one particle entry */
13
14
     MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
15
     MPI_Type_commit(&Onepair);
16
     MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);
17
18
19
     Example 4.18 The same manipulations as in the previous example, but use absolute
20
     addresses in datatypes.
21
22
     struct Partstruct
23
     {
24
         int
                 type;
25
         double d[6];
26
         char
                b[7];
27
     };
28
29
     struct Partstruct particle[1000];
30
^{31}
                 /* build datatype describing first array entry */
32
33
     MPI_Datatype Particletype;
34
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
35
     int
                   block[3] = \{1, 6, 7\};
36
                   disp[3];
     MPI_Aint
37
38
     MPI_Get_address(particle, disp);
39
     MPI_Get_address(particle[0].d, disp+1);
40
     MPI_Get_address(particle[0].b, disp+2);
41
     MPI_Type_create_struct(3, block, disp, type, &Particletype);
42
43
     /* Particletype describes first array entry -- using absolute
44
        addresses */
45
46
                        /* 5.1:
47
                  send the entire array */
48
```

```
\mathbf{2}
MPI_Type_commit(&Particletype);
                                                                                        3
MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
                                                                                        4
                                                                                        5
                   /* 5.2:
                                                                                        6
          send the entries of type zero,
                                                                                        7
                                                                                        8
          preceded by the number of such entries */
                                                                                        9
                                                                                        10
MPI_Datatype Zparticles, Ztype;
                                                                                        11
int
              zdisp[1000];
                                                                                        12
              zblock[1000], i, j, k;
                                                                                        13
int
                                                                                        14
int
              zzblock[2] = {1,1};
                                                                                        15
MPI_Datatype zztype[2];
                                                                                        16
MPI_Aint
              zzdisp[2];
                                                                                        17
                                                                                        18
j=0;
for (i=0; i < 1000; i++)
                                                                                        19
                                                                                        20
    if (particle[i].type == 0)
                                                                                        21
         {
             for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
                                                                                        22
                                                                                        23
             zdisp[j] = i;
                                                                                        ^{24}
             zblock[j] = k-i;
                                                                                        25
             j++;
                                                                                        26
             i = k;
         }
                                                                                        27
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
                                                                                        28
                                                                                        29
/* Zparticles describe particles with type zero, using
                                                                                        30
   their absolute addresses*/
                                                                                        ^{31}
/* prepend particle count */
                                                                                        32
                                                                                        33
MPI_Get_address(&j, zzdisp);
                                                                                        34
zzdisp[1] = (MPI_Aint)0;
zztype[0] = MPI_INT;
                                                                                        35
zztype[1] = Zparticles;
                                                                                        36
                                                                                        37
MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
                                                                                        38
                                                                                        39
MPI_Type_commit(&Ztype);
MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
                                                                                        40
                                                                                        41
                                                                                        42
                                                                                        43
Example 4.19 Handling of unions.
                                                                                        44
                                                                                        45
union {
                                                                                        46
   int
            ival;
                                                                                        47
   float
            fval;
                                                                                        48
```

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

}

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

```
4.2
            Pack and Unpack
1
\mathbf{2}
     Some existing communication libraries provide pack/unpack functions for sending noncon-
3
     tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending
4
     it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are
5
     described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking.
6
     The user specifies the layout of the data to be sent or received, and the communication
7
     library directly accesses a noncontiguous buffer. The pack/unpack routines are provided
8
     for compatibility with previous libraries. Also, they provide some functionality that is not
9
     otherwise available in MPI. For instance, a message can be received in several parts, where
10
     the receive operation done on a later part may depend on the content of a former part.
11
     Another use is that outgoing messages may be explicitly buffered in user supplied space,
12
     thus overriding the system buffering policy. Finally, the availability of pack and unpack
13
     operations facilitates the development of additional communication libraries layered on top
14
     of MPI.
15
16
17
     MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)
18
       IN
                 inbuf
                                              input buffer start (choice)
19
20
       IN
                 incount
                                               number of input data items (non-negative integer)
21
                                              datatype of each input data item (handle)
       IN
                 datatype
22
       OUT
                 outbuf
                                              output buffer start (choice)
23
^{24}
       IN
                 outsize
                                              output buffer size, in bytes (non-negative integer)
25
       INOUT
                 position
                                              current position in buffer, in bytes (integer)
26
       IN
                                               communicator for packed message (handle)
                 comm
27
28
29
     int MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype,
30
                     void *outbuf, int outsize, int *position, MPI_Comm comm)
^{31}
     MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
32
          TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
33
          TYPE(*), DIMENSION(..) ::
                                           outbuf
34
          INTEGER, INTENT(IN) :: incount, outsize
35
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
          INTEGER, INTENT(INOUT) :: position
37
          TYPE(MPI_Comm), INTENT(IN) :: comm
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
41
          <type> INBUF(*), OUTBUF(*)
42
          INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
43
          Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer
44
     space specified by outbuf and outsize. The input buffer can be any communication buffer
45
     allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize
46
     bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were
47
     a communication buffer for a message of type MPI_PACKED).
48
```

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.

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm) inbuf IN input buffer start (choice) IN insize size of input buffer, in bytes (non-negative integer) INOUT position current position in bytes (integer) OUT outbuf output buffer start (choice) IN outcount number of items to be unpacked (integer) IN datatype datatype of each output data item (handle) IN communicator for packed message (handle) comm int MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf, 20int outcount, MPI_Datatype datatype, MPI_Comm comm) 21MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, 22 ierror) 23TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf TYPE(*), DIMENSION(..) :: outbuf 25INTEGER, INTENT(IN) :: insize, outcount INTEGER, INTENT(INOUT) :: position 27TYPE(MPI_Datatype), INTENT(IN) :: datatype 28 TYPE(MPI_Comm), INTENT(IN) :: comm 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR) <type> INBUF(*), OUTBUF(*) INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the 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 buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of **position** is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

Note the difference between MPI_RECV and MPI_UNPACK: in Advice to users. 45MPI_RECV, the count argument specifies the maximum number of items that can 46 be received. The actual number of items received is determined by the length of 47the incoming message. In MPI_UNPACK, the count argument specifies the actual 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

56 7

8

9

10

11

12

13

14

15

1617

18 19

 24

26

31

32

33

34

35

36

37

38

39

40

41

42

number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data 8 9 part of a message as being the sequence obtained by concatenating the successive values sent 10 in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, 11as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or 12sscanf in C, for a similar function.) 13

Several messages can be successively packed into one **packing unit**. This is effected 14by several successive related calls to MPI_PACK, where the first call provides position = 0, 1516and each successive call inputs the value of position that was output by the previous call, 17and 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 18 send buffer that is the "concatenation" of the individual send buffers. 19

A packing unit can be sent using type MPI_PACKED. Any point to point or collective 2021communication 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 22operation, with any datatype: the type matching rules are relaxed for messages sent with 23 24 type MPI_PACKED.

25A message sent with any type (including MPI_PACKED) can be received using the type 26MPI_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 27several successive messages. This is effected by several successive related calls to 28

MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the 29value of position that was output by the previous call, and the same values for inbuf, insize 30 and comm. 31

32 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 33 34packing 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 35 sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of 36 related unpack calls. 37

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

The following call allows the user to find out how much space is needed to pack a 44message and, thus, manage space allocation for buffers. 45

38

39

40

41

4243

- 4647
- 48

1

2

3

4

5

6

MPI_PACK_SIZE(incount, datatype, comm, size)

MPI_PACK	_SIZE(Incount, datatype, com	n, size <i>j</i>	1	
IN	incount	count argument to packing call (non-negative integer)	2 3	
IN	datatype	datatype argument to packing call (handle)	3	
IN	comm	communicator argument to packing call (handle)	5	
		· · · · · · · · · · · · · · · · · ·	6	
OUT	size	upper bound on size of packed message, in bytes (non- negative integer)	7	
		negative mieger)	8	
int MDT D	ack size(int incount MPI	_Datatype datatype, MPI_Comm comm,	9	
IIIC THII_I	int *size)	_batatype datatype, M1_comm comm,	10	
			11	
	size(incount, datatype, c		12	
	ER, INTENT(IN) :: incour		13 14	
	MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) ::		14	
	ER, INTENT(OUT) :: size	comm	16	
	ER, OPTIONAL, INTENT(OUT)	:: ierror	17	
			18	
	SIZE(INCOUNT, DATATYPE, C		19	
INTEG	ER INCOUNT, DATATYPE, COM	M, SIZE, IERRUR	20	
A call	to MPI_PACK_SIZE(incount, d	atatype, comm, size) returns in size an upper bound	21	
	· ·	${ m ed}$ by a call to MPI_PACK(inbuf, incount, datatype,	22 23	
outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed				
by the size	parameter, then MPI_PACK_	SIZE sets the value of size to MPI_UNDEFINED.	24 25	
Ratio	<i>nale.</i> The call returns an up	per bound, rather than an exact bound, since the	25 26	
	-	ack the message may depend on the context (e.g.,	27	
	· ·	unit may take more space). (End of rationale.)	28	
			29	
Example	4.21 An example using MPI	ΡΔΓΚ	30	
-			31	
int	<pre>position, i, j, a[2];</pre>		32	
char	buff[1000];		33	
MDT Comm	rank(MPI_COMM_WORLD, &myr	nank).	34	
if (myran)		alik),	35	
{			36 37	
			51	

```
position = 0;
MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
MPI_Send(buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else /* RECEIVER CODE */
MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Example 4.22 An elaborate example.

/* SENDER CODE */

1

38 39

40

 41

42

43

44

45

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

```
/* Unpack i */
position = 0;
MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
/* Unpack a[0]...a[i-1] */
MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

Example 4.23 Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

```
14
int count, gsize, counts[64], totalcount, k1, k2, k,
                                                                                     15
     displs[64], position, concat_pos;
                                                                                     16
char chr[100], *lbuf, *rbuf, *cbuf;
                                                                                     17
                                                                                     18
MPI_Comm_size(comm, &gsize);
                                                                                    19
MPI_Comm_rank(comm, &myrank);
                                                                                    20
      /* allocate local pack buffer */
                                                                                    21
MPI_Pack_size(1, MPI_INT, comm, &k1);
                                                                                    22
                                                                                    23
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
                                                                                    24
k = k1+k2;
                                                                                    25
lbuf = (char *)malloc(k);
                                                                                     26
      /* pack count, followed by count characters */
                                                                                    27
                                                                                    28
position = 0;
                                                                                    29
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
                                                                                    30
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
                                                                                    31
if (myrank != root) {
                                                                                    32
                                                                                    33
    /* gather at root sizes of all packed messages */
                                                                                    34
    MPI_Gather(&position, 1, MPI_INT, NULL, 0,
                MPI_DATATYPE_NULL, root, comm);
                                                                                    35
                                                                                    36
                                                                                    37
    /* gather at root packed messages */
                                                                                    38
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                                                                                    39
                 NULL, NULL, MPI_DATATYPE_NULL, root, comm);
                                                                                     40
                                                                                    41
} else {
           /* root code */
                                                                                    42
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
                                                                                    43
                                                                                    44
                MPI_INT, root, comm);
                                                                                     45
                                                                                     46
    /* gather all packed messages */
                                                                                     47
    displs[0] = 0;
                                                                                     48
    for (i=1; i < gsize; i++)</pre>
```

Unofficial Draft for Comment Only

1 2

3

4

5 6 7

8

9 10 11

12

```
displs[i] = displs[i-1] + counts[i-1];
         totalcount = displs[gsize-1] + counts[gsize-1];
         rbuf = (char *)malloc(totalcount);
         cbuf = (char *)malloc(totalcount);
         MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
                      counts, displs, MPI_PACKED, root, comm);
         /* unpack all messages and concatenate strings */
         concat_pos = 0;
10
         for (i=0; i < gsize; i++) {</pre>
             position = 0;
12
             MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
13
                         &position, &count, 1, MPI_INT, comm);
14
             MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
15
                         &position, cbuf+concat_pos, count, MPI_CHAR, comm);
16
             concat_pos += count;
17
         }
         cbuf[concat_pos] = '\0';
19
     }
20
```

Canonical MPI PACK and MPI UNPACK 4.3

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.*)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7 8

9

11

18

21

22 23

24

25

26

2728

29

30

 31

32 33

34

35

36

MPI_PACK	<pre>C_EXTERNAL(datarep, inbuf, i</pre>	incount, datatype, outbuf, outsize, position)	1		
IN	datarep	data representation (string)	2 3		
IN	inbuf	input buffer start (choice)	3 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_P		<pre>datarep[], const void *inbuf, int incount, be, void *outbuf, MPI_Aint outsize,</pre>	12 13 14 15		
MPI_Pack_	external(datarep, inbuf,	incount, datatype, outbuf, outsize,	16		
	position, ierror)		17		
	ACTER(LEN=*), INTENT(IN) (*), DIMENSION(), INTEN	1	18 19		
	(*), DIMENSION(), INTEN		20		
	ER, INTENT(IN) :: incou		21		
	MPI_Datatype), INTENT(IN	V 1	22		
), INTENT(IN) :: outsize), INTENT(INOUT) :: position	23 24		
	ER, OPTIONAL, INTENT(OUT)	-	25		
		INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	26		
In 1_1 AON_	POSITION, IERROR)	INCOUNT, DATATILE, COTDOF, COTSIZE,	27		
INTEG	ER INCOUNT, DATATYPE, IE	RROR	28 29		
	ER(KIND=MPI_ADDRESS_KIND)) OUTSIZE, POSITION	30		
	ACTER*(*) DATAREP >> INBUF(*), OUTBUF(*)		31		
Coppe	> INDOI (*), 001D01 (*)		32		
			33 34		
MPI_UNPA	ACK_EXTERNAL(datarep, inbu	uf, insize, position, outbuf, outsize, position)	35		
IN	datarep	data representation (string)	36		
IN	inbuf	input buffer start (choice)	37		
IN	insize	input buffer size, in bytes (integer)	38 39		
INOUT	position	current position in buffer, in bytes (integer)	40		
OUT	outbuf	output buffer start (choice)	41		
IN	outcount	number of output data items (integer)	42 43		
IN	datatype	datatype of output data item (handle)	43 44		
			45		
<pre>int MPI_Unpack_external(const char datarep[], const void *inbuf,</pre>					
		_Aint *position, void *outbuf,	47 48		
	int outcount, MPI_Da	tatype datatype)	48		

```
1
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
\mathbf{2}
                   datatype, ierror)
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
                                           datatype of each input data item (handle)
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)
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 5.14.

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.

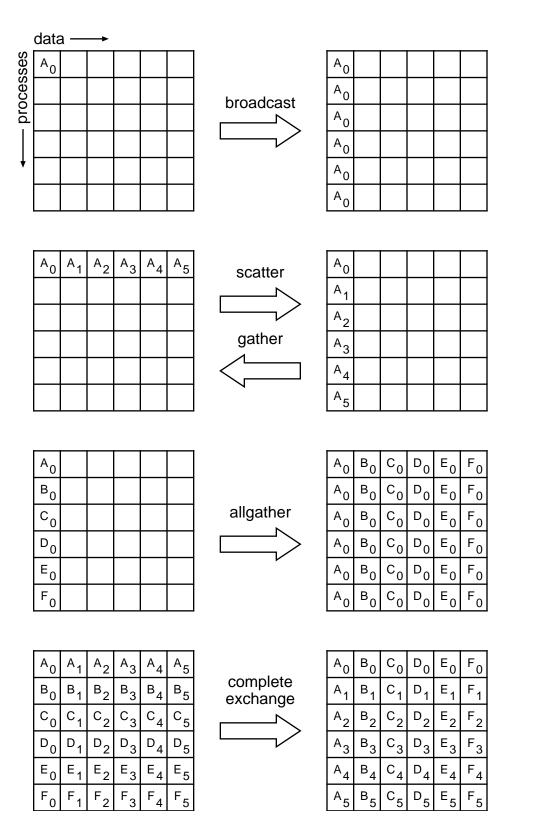


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

 $\mathbf{2}$

 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 5.14. (*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 5.14. (*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, [56]:

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

 MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV 	
 MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW 	
 MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER 	
MPI_BARRIER, MPI_IBARRIER	
All-To-One All processes contribute to the result. One process receives the result.	

MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV

MPI_REDUCE, MPI_IREDUCE

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

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

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

MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN

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

The application of collective communication to intercommunicators is best described 35in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be 36 described as collecting data from all members of one group with the result appearing in all 37 members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the 39 other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.

The following collective operations also apply to intercommunicators:

- MPI_BARRIER, MPI_IBARRIER
- MPI_BCAST, MPI_IBCAST

Unofficial Draft for Comment Only

1

2

3 4

5

6

7

8 9

21

22 23

 24

2526

27

28 29

30 31

32

33

34

38

40

41

42

43

44

4546

47

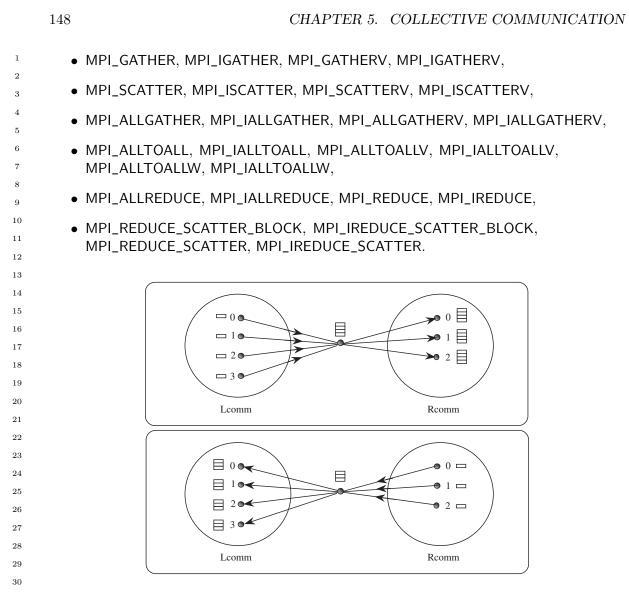


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

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

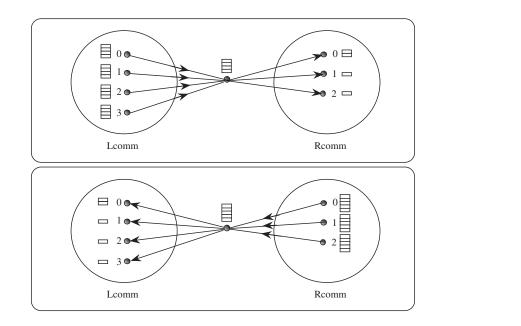


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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. $44 \\ 45$

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)
```

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 IERROR

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.

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

1

 $\mathbf{2}$

3

4

5 6

20 21

22

23

24

25

26

in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1

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

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

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

5.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

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

int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Unofficial Draft for Comment Only

 24

1 2	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4 5	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
6	ROOT, COMM, IERROR)
7	<type> SENDBUF(*), RECVBUF(*)</type>
8	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
9	If comm is an intracommunicator, each process (root process included) sends the con-
10 11	tents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is $as if$ each of the n processes in the group (including
12	the root process) had executed a call to
13	
14	MPI_Send(sendbuf, sendcount, sendtype, root ,),
15 16	and the root had executed n calls to
17	
18	MPI_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,),
19	where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent.
20	An alternative description is that the n messages sent by the processes in the group
21 22	are concatenated in rank order, and the resulting message is received by the root as if by a
23	call to MPI_RECV(recvbuf, recvcount·n, recvtype,).
24	The receive buffer is ignored for all non-root processes.
25	General, derived datatypes are allowed for both sendtype and recvtype. The type signa- 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
27 28	of data received, pairwise between each process and the root. Distinct type maps between
28 29	sender and receiver are still allowed.
30	All arguments to the function are significant on process root, while on other processes,
31	only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments 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 34	written more than once. Such a call is erroneous.
35	Note that the recvcount argument at the root indicates the number of items it receives
36	from each process, not the total number of items it receives.
37	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
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 41	If comm is an intercommunicator, then the call involves all processes in the intercom-
42	municator, but with one group (group A) defining the root process. All processes in the
43	other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root pages the value MPI BOOT in root. All other proceedings in group A.
44	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
45	the root. The send buffer arguments of the processes in group B must be consistent with
46 47	the receive buffer argument of the root.
48	

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, ¹ comm)					
IN	sendbuf	starting address of and huffer (shairs)	3		
		starting address of send buffer (choice)	4		
IN	sendcount	number of elements in send buffer (non-negative integer)	5 6		
IN	sendtype	data type of send buffer elements (handle)	7		
OUT	recvbuf	address of receive buffer (choice, significant only at root)	8 9 10		
IN	recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process (significant only at root)	11 12 13		
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)	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		
<pre>int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>					
MPT Gather	v(sendbuf, sendcount, se	endtype, recvbuf, recvcounts, displs,	27 28		
	recvtype, root, comm		29		
TYPE(*), DIMENSION(), INTENT		30		
TYPE(*), DIMENSION() :: re	ecvbuf	31		
		<pre>ount, recvcounts(*), displs(*), root</pre>	32		
	<pre>IPI_Datatype), INTENT(IN)</pre>		33		
	<pre>IPI_Comm), INTENT(IN) ::</pre>		34		
INTEGE	R, OPTIONAL, INTENT(OUT)	:: lerror	35		
MPI_GATHER	RV(SENDBUF, SENDCOUNT, SE	NDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	36		
	RECVTYPE, ROOT, COMM	, IERROR)	37 38		
• -	<pre>SENDBUF(*), RECVBUF(*)</pre>		39		
INTEGE		<pre>RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,</pre>	40		
COMM, IERROR 41					
MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count 42					
of data from each process, since recvcounts is now an array. It also allows more flexibility 43					
	as to where the data is placed on the root, by providing the new argument, displs. 44				
If comm is an intracommunicator, the outcome is $as if$ each process, including the root 45					

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

process, sends a message to the root,

Unofficial Draft for Comment Only

4647

1	and the root executes n receives,
2 3	MPI_Recv(recvbuf+displs[j]· extent(recvtype), recvcounts[j], recvtype, i,).
4 5 6	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).
7 8	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
9 10 11	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
12	in Example 5.6.
13 14	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.
15 16 17	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.
18 19	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
20 21	place in the receive buffer.
21	If comm is an intercommunicator, then the call involves all processes in the intercom-
23 24 25 26 27 28 29	municator, 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.
30	5.5.1 Examples using MPI_GATHER, MPI_GATHERV
31 32	The examples in this section use intracommunicators.
33	Example 5.2
34 35	Gather 100 ints from every process in group to root. See Figure 5.4.
36	MPI_Comm comm;
37	<pre>int gsize,sendarray[100];</pre>
38 39	<pre>int root, *rbuf;</pre>
40	<pre>MPI_Comm_size(comm, &gsize);</pre>
41 42 43	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int)); MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);</pre>
$44 \\ 45$	Example 5.3
46 47	Previous example modified — only the root allocates memory for the receive buffer.
48	

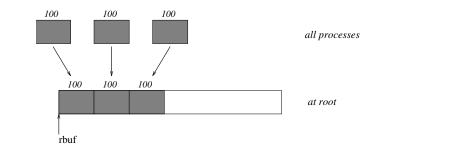


Figure 5.4: The root process gathers 100 ints from each process in the group.

```
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>
```

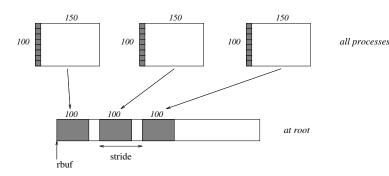


Figure 5.6: The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

```
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 100×150 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

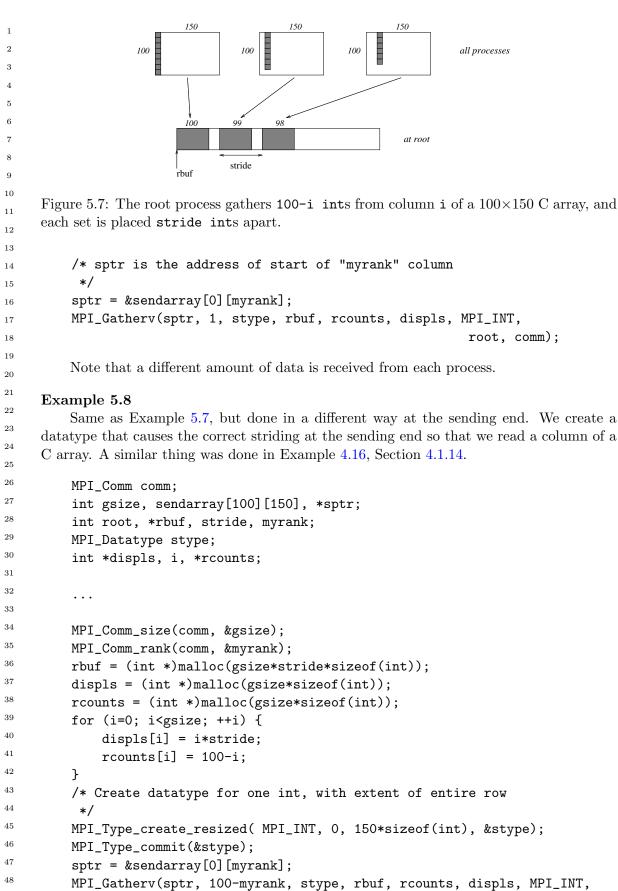
21

22 23 24

25

26

```
CHAPTER 5. COLLECTIVE COMMUNICATION
```



2 3

4

5

6 7

 $45 \\ 46$

47 48

root, comm);

```
Example 5.9
```

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

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

Example 5.10

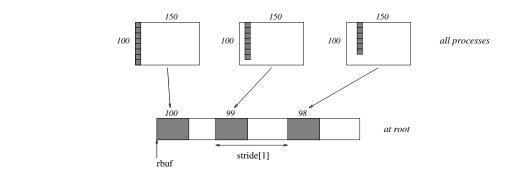


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

18

1

2

7

8

9 10

11

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

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

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

3 4 MPI_Recv(recvbuf, recvcount, recvtype, i,...).

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount 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.

5 6

17

18

19

20 21 The send buffer is ignored for all non-root processes.

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

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

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

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

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

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

- 34
- 35 36
- 37
- 38 39
- 40
- 41
- 42 43
- 44
- 45
- 46
- 47
- 48

MPI_SCAT	TERV(sendbuf, sendcounts, d	ispls, sendtype, recvbuf, recvcount, recvtype, root,	1		
	comm)		2		
IN	sendbuf	address of send buffer (choice, significant only at root)	3 4		
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	4 5 6		
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i	7 8 9		
IN	sendtype	data type of send buffer elements (handle)	10 11		
OUT	recvbuf	address of receive buffer (choice)	12		
IN	recvcount	number of elements in receive buffer (non-negative in- teger)	13 14		
IN	recvtype	data type of receive buffer elements (handle)	15		
IN	root	rank of sending process (integer)	16 17		
IN	comm	communicator (handle)	18		
			19		
int MPI_So	catterv(const void* send)	ouf, const int sendcounts[],	20		
		MPI_Datatype sendtype, void* recvbuf,	21 22		
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)				
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, 24					
recvtype, root, comm, ierror) 25					
			26		
			27		
		<pre>ounts(*), displs(*), recvcount, root</pre>	28		
	<pre>MPI_Datatype), INTENT(IN)</pre>	VI VI	29		
	<pre>MPI_Comm), INTENT(IN) ::</pre>		30		
INTEGE	ER, OPTIONAL, INTENT(OUT)) :: ierror	31		
MPI_SCATTE	ERV(SENDBUF, SENDCOUNTS,	DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	32		
	RECVTYPE, ROOT, COMM	, IERROR)	33		
• -	<pre>SENDBUF(*), RECVBUF(*)</pre>		$\frac{34}{35}$		
INTEGE		(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	36		
	COMM, IERROR		37		
MPI_S	CATTERV is the inverse oper	ation to MPI_GATHERV.	38		
MPI_S	CATTERV extends the funct	ionality of MPI_SCATTER by allowing a varying	39		
	-	s, since sendcounts is now an array. It also allows	40		
	more flexibility as to where the data is taken from on the root, by providing an additional 41				
0 ,	argument, displs. 42				
	If comm is an intracommunicator, the outcome is as if the root executed n send oper-				
ations, 44					

MPI_Send(sendbuf+displs[i]· extent(sendtype), sendcounts[i], sendtype, i,...), and each process executed a receive, 45

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

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

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

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

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

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

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

26 27

28

30

 31

32

33

48

5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

²⁹ The examples in this section use intracommunicators.

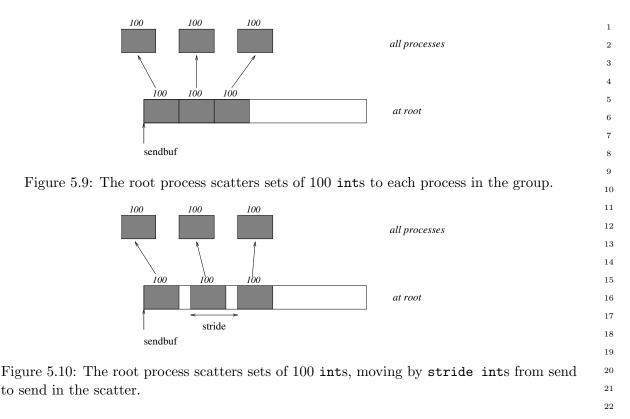
Example 5.11

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

34MPI_Comm comm; 35 int gsize,*sendbuf; 36 int root, rbuf[100]; 37 . . . 38 MPI_Comm_size(comm, &gsize); 39 sendbuf = (int *)malloc(gsize*100*sizeof(int)); 40 . . . 41 MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm); 4243

⁴⁴ Example 5.12

The reverse of Example 5.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of MPI_SCATTERV. Assume *stride* \geq 100. See Figure 5.10.



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

 24

25

26 27

28 29

30

 31

32

33

34

35

36

37

38

39

40 41

42

43

44

 $45 \\ 46$

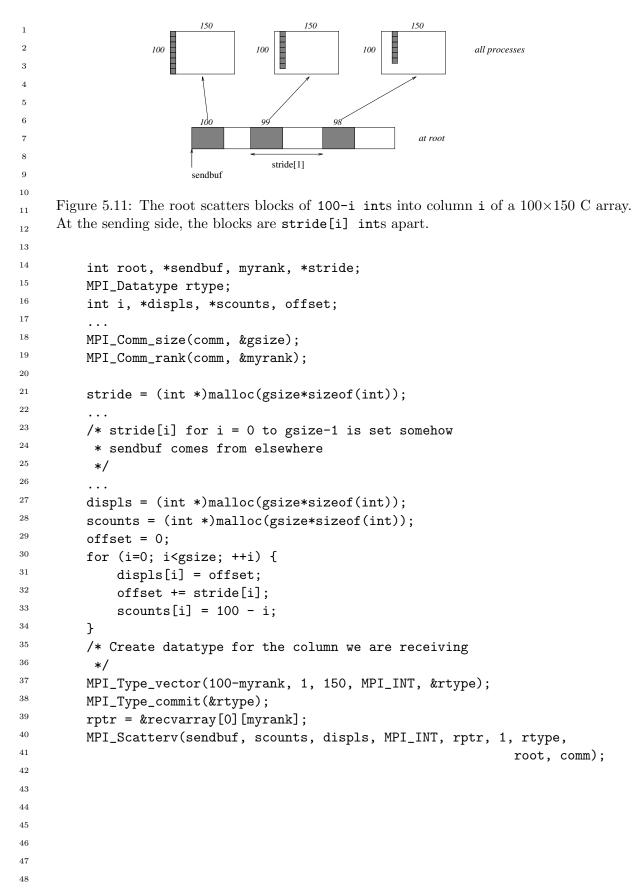
47

48

Example 5.13

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

MPI_Comm comm; int gsize,recvarray[100][150],*rptr;



5.7 Gather-to-all

		ount, sendtype, recvbuf, recvcount, recvtype, comm)
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
nt MPI.	MPI_Datatype se	l* sendbuf, int sendcount, endtype, void* recvbuf, int recvcount, ecvtype, MPI_Comm comm)
TYPI TYPI INTI TYPI TYPI	<pre>comm, ierror) E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) ::</pre>	sendcount, recvcount ENT(IN) :: sendtype, recvtype EN) :: comm
<ty]< td=""><td>COMM, IERROR) pe> SENDBUF(*), RECVE</td><td>COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, BUF(*) TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</td></ty]<>	COMM, IERROR) pe> SENDBUF(*), RECVE	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, BUF(*) TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
he resul by every The he type If co	t, instead of just the roo process and placed in the type signature associate signature associated wit	bught of as MPI_GATHER, but where all processes receive t. The block of data sent from the j-th process is received the j-th block of the buffer recvbuf. d with sendcount, sendtype, at a process must be equal to h recvcount, recvtype at any other process. eator, the outcome of a call to MPI_ALLGATHER() is as
MPI_(Gather(sendbuf,sendco	ount,sendtype,recvbuf,recvcount, recvtype,root,comm)
	= 0 ,, n-1. The r corresponding rules for	ules for correct usage of MPI_ALLGATHER are easily found
		tracommunicators is specified by passing the value

The "in place" option for intracommunicators is specified by passing the value 47 MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. 48

Unofficial Draft for Comment Only

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.

³ If comm is an intercommunicator, then each process of one group (group A) contributes ⁴ sendcount data items; these data are concatenated and the result is stored at each process ⁵ in the other group (group B). Conversely the concatenation of the contributions of the ⁶ processes in group B is stored at each process in group A. The send buffer arguments in ⁷ group A must be consistent with the receive buffer arguments in group B, and vice versa.

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

17 18 19

20 21

8 9

10

11

12

13

14

15

16

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

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

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

5

CHAPTER 5. COLLECTIVE COMMUNICATION

5.8 All-to-All Scatter/Gather

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

6	IN	sendbuf	starting address of send buffer (choice)
7 8	IN	sendcount	number of elements sent to each process (non-negative integer)
9 10	IN	sendtype	data type of send buffer elements (handle)
11	OUT	recvbuf	address of receive buffer (choice)
12 13	IN	recvcount	number of elements received from any process (non-negative integer)
14 15	IN	recvtype	data type of receive buffer elements (handle)
16	IN	comm	communicator (handle)
17 18 19 20 21	int MPI_Al		ouf, int sendcount, MPI_Datatype sendtype, ecvcount, MPI_Datatype recvtype,
22 23 24 25 26 27 28 29	TYPE(* TYPE(* INTEGE TYPE(M TYPE(M	<pre>ill(sendbuf, sendcount, s comm, ierror) ;), DIMENSION(), INTENT ;), DIMENSION() :: re cR, INTENT(IN) :: sendco iPI_Datatype), INTENT(IN) iPI_Comm), INTENT(IN) :: cR, OPTIONAL, INTENT(OUT)</pre>	ecvbuf ount, recvcount :: sendtype, recvtype comm
30 31 32 33 34	<type></type>	COMM, IERROR) • SENDBUF(*), RECVBUF(*)	SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
35 36 37 38 39 40 41 42 43	sends distin by process j The typ the type sig that the amo every pair o If comm	ct data to each of the receiv and is placed in the i-th blo pe signature associated with gnature associated with recvo ount of data sent must be equ of processes. As usual, however	sendcount, sendtype, at a process must be equal to count, recvtype at any other process. This implies al to the amount of data received, pairwise between er, the type maps may be different. ne outcome is as if each process executed a send to
44 45 46			nt(sendtype),sendcount,sendtype,i,),
47	and a r	eceive from every other proc	ess with a call to,
48	MPI_Rec	v(recvbuf+i∙ recvcount∙ exten	t(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.*)

Unofficial Draft for Comment Only

 $\mathbf{2}$

12	MPI_ALLT(OALLV(sendbuf, sendcounts, s 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)
24 25 26 27 28 29 30 31 32 33 33 34	<pre>const int sdispls[], MP1_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MP1_Datatype recvtype, MP1_Comm comm) MP1_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, ierror) TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) TYPE(*) D is to be a sended.</pre>		
35 36		MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)	
37 38 39 40 41 42 43 44 45 46 47	<type INTEG MPI_A the send is side is spec If com</type 	RDISPLS, RECVTYPE, C > SENDBUF(*), RECVBUF(*) ER SENDCOUNTS(*), SDISPLS RECVTYPE, COMM, IER ALLTOALLV adds flexibility to specified by sdispls and the lo ified by rdispls. m is an intracommunicator, t	S(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
48	size.		

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

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

MPI_Send(sendbuf+sdispls[i] · extent(sendtype),sendcounts[i],sendtype,i,...), and received a message from every other process with a call to MPI_Recv(recvbuf+rdispls[i] extent(recvtype),recvcounts[i],recvtype,i,...). All arguments on all processes are significant. The argument comm must have identical values on all processes. The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts array and the recvtype, and is taken from the locations of the receive buffer specified by rdispls. Specifying the "in place" option (which must be given on all Advice to users. processes) implies that the same amount and type of data is sent and received between

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

 31

1 2	MPI_ALLT	OALLW(sendbuf, sendcounts, s recvtypes, comm)	displs, 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 spec- ifies the type of data to send to process j (array of handles)
14	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 23	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
23 24 25 26	IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)
27	IN	comm	communicator (handle)
28 29 30 31 32 33		const int sdispls[], void* recvbuf, const const MPI_Datatype r	<pre>dbuf, const int sendcounts[], const MPI_Datatype sendtypes[], int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm) , sdispls, sendtypes, recvbuf, recvcounts,</pre>
34 35 36 37 38 39 40 41 42 43 44 45 46	TYPE(TYPE(INTEG TYPE(TYPE(TYPE(INTEG MPI_ALLTO	<pre>rdispls, recvtypes, *), DIMENSION(), INTENT *), DIMENSION() :: re ER, INTENT(IN) :: sendco rdispls(*) MPI_Datatype), INTENT(IN) MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)</pre>	<pre>comm, ierror) T(IN) :: sendbuf ecvbuf ounts(*), sdispls(*), recvcounts(*),) :: sendtypes(*)) :: recvtypes(*) comm) :: ierror , SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,</pre>
47 48	INTEG	ER SENDCOUNTS(*), SDISPLS RDISPLS(*), RECVTYPP	S(*), SENDTYPES(*), RECVCOUNTS(*), ES(*), COMM, IERROR

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.

 $46 \\ 47$

 $\mathbf{2}$

 24

 31

```
176
                                           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
7
                  recvbuf
                                              address of receive buffer (choice, significant only at
8
                                              root)
9
       IN
                  count
                                              number of elements in send buffer (non-negative inte-
10
                                              ger)
11
                                              data type of elements of send buffer (handle)
       IN
                  datatype
12
       IN
                                              reduce operation (handle)
13
                  ор
14
       IN
                                              rank of root process (integer)
                  root
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
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
22
          TYPE(*), DIMENSION(..) :: recvbuf
23
          INTEGER, INTENT(IN) :: count, root
^{24}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
29
          <type> SENDBUF(*), RECVBUF(*)
30
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
^{31}
32
          If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the
33
     input buffer of each process in the group, using the operation op, and returns the combined
34
      value in the output buffer of the process with rank root. The input buffer is defined by
35
      the arguments sendbuf, count and datatype; the output buffer is defined by the arguments
36
      recvbuf, count and datatype; both have the same number of elements, with the same type.
37
      The routine is called by all group members using the same arguments for count, datatype, op,
38
      root and comm. Thus, all processes provide input buffers of the same length, with elements
39
      of the same type as the output buffer at the root. Each process can provide one element, or a
40
      sequence of elements, in which case the combine operation is executed element-wise on each
41
      entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains
42
      two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then
43
      recvbuf(1) = global \max(sendbuf(1)) and recvbuf(2) = global \max(sendbuf(2)).
44
          Section 5.9.2, lists the set of predefined operations provided by MPI. That section also
45
      enumerates the datatypes to which each operation can be applied.
46
          In addition, users may define their own operations that can be overloaded to operate
47
      on several datatypes, either basic or derived. This is further explained in Section 5.9.5.
48
```

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

5.9.2 Predefined Reduction Operations The following predefined operations are supplied for MPI_REDUCE and related functions 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. These operations are invoked by placing the following in op. Name Meaning MPI_MAX maximum MPI_SUM sum MPI_LAND logical and MPI_LOR logical and MPI_BOR bit-wise and MPI_LOR logical or MPI_BOR bit-wise or MPI_BOR bit-wise or MPI_MAXLOC max value and location MPI_MALOC min value and location MPI_MINLOC min value and location MPI_UNSIGNED_LONG, MPI_SHORT, MPI_UNSIGNED, MPI_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_SHORT, MPI_UNSIGNED_LONG, MPI_SHORT, MPI_UNSIGNED_LONG, MPI_SIGNED_LONG, MPI_UNSIGNED_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_LONG, MPI_SHORT, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNTEGER, and handler returned from MPI_NTEGER4, MPI_INTEGER4, MPI_INTEGER4		178 CHA	PTER 5. COLLECTIVE COMMUNICATION	
3 The following predefined operations are supplied for MPL_REDUCE and related functions 4 MPI_SCAN, MPL_EXSCAN, all nonblocking variants of those (see Section 5.12), and 6 MPI_SCAN, MPL_EXSCAN, all nonblocking variants of those (see Section 5.12), and 7 MPI_REDUCE_LOCAL. These operations are invoked by placing the following in op. 7 Meaning 7 MPI_MAX 7 maximum 7 MPI_MAX 7 maximum 7 MPI_MAX 7 maximum 7 MPI_MAX 7 maximum 7 maximum 7 MPI_MAX 7 maximum 7 maximum 7 MPI_SOM 7 moduct 7 MPI_SON 7 moduct 7 moduct 7 MPI_LOR 7 logical and 7 moduct 7 MPI_LOR 8 bit-wise exclusive or (xor) 9 MPI_LXOR bibic-wise exclusive or (xor) 9 <th></th> <th colspan="3">5.9.2 Predefined Reduction Operations</th>		5.9.2 Predefined Reduction Operations		
Name Meaning MPI_MAX maximum MPI_MIN minimum MPI_SUM sum MPI_PROD product MPI_BAND bit-wise and MPI_LOR logical or MPI_BOR bit-wise or MPI_BOR bit-wise or MPI_BOR bit-wise or (xor) MPI_BXOR logical exclusive or (xor) MPI_MAXLOC max value and location The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Sec- tions of op and datatype arguments. First, define groups of MPI basic datatypes in the following way. For the other predefined operations, we enumerate below the allowed combi- c integer: MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_SHORT, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG, INT, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG (as synonym), MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNTEGER, MPI_UNTS_T, MPI_UNT16_T, MPI_INTEGER, MPI_UNTS_T, MPI_UNT16_T, MPI_INTEGER, mPI_INTEGER and handles returned from MPI_ITYPE_CREATE_F90_INTEGER, and, if available, MPI_INTEGER4, MPI_INTEGER8, and MPI_INTEGER4, MPI_INTEGER8, and MPI_INTEGER4	3 4 5 6	IPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, IPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and		
MPL_MAX maximum MPL_MIN minimum MPL_SUM sum MPL_FROD product MPL_AND logical and MPL_BAND bit-wise and MPL_BOR bit-wise or MPL_KOR logical or MPL_KOR logical exclusive or (xor) MPL_BOR bit-wise exclusive or (xor) MPL_BAND bit-wise exclusive or (xor) MPL_MAXLOC max value and location MPL_MINLOC min value and location The two operations MPL_MINLOC and MPL_MAXLOC are discussed separately in Sec- tion 5.9.4. For the other predefined operations, we enumerate below the allowed combi- nations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way. C C C Integer: MPL_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_SHORT, MPL_UNSIGNED_LONG, MPI_SIGNED, MPI_MINLOK MPI_UNSIGNED_LONG, MPI_SIGNED, MPI_MINLOK MPI_UNSIGNED_LONG, MPI_SIGNED, MPI_MINLOK MPI_UNSIGNED_LONG, MPI_UNSIGNED, MPI_MINLOK MPI_UNSIGNED_LONG, MPI_SIGNED, MPI_MINLOK MPI_UNSIGNED_LONG, MPI_NTEGER MPI_UNT8_T, MPI_UNT6_T, MPI_UNT64_T MPI_NTEGER MPI_INTEGER MPI_INTEGER MPI_		Name	Meaning	
 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_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_UNT8_T, MPI_UNT16_T, MPI_UINT8_T, MPI_UINT64_T, MPI_INTEGER And handles returned from MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER1, MPI_INTEGER3, and MPI_INTEGER4, MPI_INTEGER8, and MPI_INTEGER16 	 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 	MPI_MIN MPI_SUM MPI_PROD MPI_LAND MPI_BAND MPI_BOR MPI_BOR MPI_LXOR MPI_BXOR MPI_MAXLOC MPI_MINLOC The two operations MPI_MINLOC and tion 5.9.4. For the other predefined oper nations of op and datatype arguments. Fin	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 MPI_MAXLOC are discussed separately in Sec- ations, we enumerate below the allowed combi-	
⁴⁶ Hoating point: MPI_FLOAT, MPI_DOUBLE, MPI_REAL, ⁴⁷ MPI_DOUBLE_PRECISION, ⁴⁷ MPI_LONG_DOUBLE,	28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46		MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, and MPI_UINT64_T MPI_INTEGER and handles returned from MPI_TYPE_CREATE_F90_INTEGER and, if available, MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER1, MPI_INTEGER3, and MPI_INTEGER16 MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION,	

	MPI_TYPE_CREATE_F90_REAL	1		
	and, if available, MPI_REAL2,	2		
	MPI_REAL4, MPI_REAL8, and MPI_REAL16	3		
Logical:	MPI_LOGICAL, MPI_C_BOOL,	4		
<u> </u>	and 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, and MPI_COMPLEX32	16		
Byte:	MPI_BYTE	17		
Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	18		
		19		
Now, the valid datatypes for each ope	eration 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 list	ed datatypes are valid in all supported program-	31		
These operations together with all listed datatypes are valid in all supported program- ming languages, see also Reduce Operations on page 686 in Section 18.2.6. The following examples use intracommunicators.				
Example 5.15				
-	duct of two vectors that are distributed across a	35 36		
A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.				
group of processes and returns the answer at node zero.				
		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-
```

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

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

47

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

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

5.9.4 MINLOC and MAXLOC

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

The operation that defines MPI_MAXLOC is:

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

where

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

and

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

MPI_MINLOC is defined similarly:

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

where

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

and

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

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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

12	according to the first component of each pair, and ties are resolved according to the second		
3	component.		
4	The reduce operation is defined to operate on arguments that consist of a pair: value and index. For both Fortran and C, types are provided to describe the pair. The potentially		
5			
6	mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented,		
7	for Fortran, by having the MPI-provided type consist of a pair of the same type as value,		
8	and coercing the index to this type also. In C, the MPI-provided pair type has distinct		
9	types and the index is an int.		
9 10	In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide		
10	a datatype argument that represents a pair (value and index). MPI provides nine such		
12	predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with		
12	each of the following datatypes.		
13	Fortran:		
14	Name	Description	
16	MPI_2REAL	pair of REALs	
	MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables	
17	MPI_2INTEGER	pair of INTEGERS	
18			
19			
20 21			
21	C:	Description	
22	Name	Description	
		float and int	
24		double and int	
25		long and int	
26	MPI_2INT	pair of int short and int	
27	MPI_SHORT_INT MPI_LONG_DOUBLE_INT	long double and int	
28	MFI_LONG_DOOBLE_INT	Tong double and Inc	
29 30	The datatype MPI_2REAL is as if defi	ned by the following (see Section 4.1).	
31	MPI_Type_contiguous(2, MPI_REAL, MP	I_2REAL);	
32			
33		EGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.	
34	The datatype MPI_SHORT_INT is as if defined by the following sequence of instructions.		
35	<pre>struct mystruct {</pre>		
36	short val;		
37	int rank;		
38	};		
39	type[0] = MPI_SHORT;		
40	<pre>type[0] = MPI_INT;</pre>		
41	disp[0] = 0;		
42	disp[0] = 0; disp[1] = offsetof(struct mystruct, rank);		
43	block[0] = 1;		
44	block[0] = 1; block[1] = 1;		
45		AD THE MOI SUCOT INT).	
46	<pre>MPI_Type_create_struct(2, block, di</pre>	sp, type, mri_onuki_iNi);	
47			
48			

```
1
Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.
                                                                                          \mathbf{2}
    The following examples use intracommunicators.
                                                                                          3
Example 5.17
                                                                                         4
    Each process has an array of 30 doubles, in C. For each of the 30 locations, compute
                                                                                          5
the value and rank of the process containing the largest value.
                                                                                          6
                                                                                          7
                                                                                          8
    /* each process has an array of 30 double: ain[30]
                                                                                          9
     */
                                                                                         10
    double ain[30], aout[30];
                                                                                         11
    int ind[30];
                                                                                         12
    struct {
                                                                                         13
        double val;
                                                                                         14
        int
               rank;
                                                                                         15
    } in[30], out[30];
                                                                                         16
    int i, myrank, root;
                                                                                         17
                                                                                         18
    MPI_Comm_rank(comm, &myrank);
                                                                                         19
    for (i=0; i<30; ++i) {
                                                                                         20
        in[i].val = ain[i];
                                                                                         21
        in[i].rank = myrank;
                                                                                         22
    }
                                                                                         23
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                         ^{24}
    /* At this point, the answer resides on process root
                                                                                         25
     */
                                                                                         26
    if (myrank == root) {
                                                                                         27
        /* read ranks out
                                                                                         28
          */
                                                                                         29
        for (i=0; i<30; ++i) {
                                                                                         30
             aout[i] = out[i].val;
                                                                                         31
             ind[i] = out[i].rank;
                                                                                         32
        }
                                                                                         33
    }
                                                                                         34
                                                                                         35
                                                                                         36
Example 5.18
                                                                                         37
    Same example, in Fortran.
                                                                                         38
                                                                                         39
    . . .
                                                                                         40
    ! each process has an array of 30 double: ain(30)
                                                                                         41
                                                                                         42
    DOUBLE PRECISION ain(30), aout(30)
    INTEGER ind(30)
                                                                                         43
                                                                                         44
    DOUBLE PRECISION in(2,30), out(2,30)
    INTEGER i, myrank, root, ierr
                                                                                         45
                                                                                         46
                                                                                         47
    CALL MPI_COMM_RANK(comm, myrank, ierr)
                                                                                         48
    DO I=1, 30
```

```
1
              in(1,i) = ain(i)
\mathbf{2}
              in(2,i) = myrank
                                    ! myrank is coerced to a double
3
         END DO
4
5
         CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
6
                                                                           comm, ierr)
7
          ! At this point, the answer resides on process root
8
9
         IF (myrank .EQ. root) THEN
10
              ! read ranks out
11
              DO I= 1, 30
12
                  aout(i) = out(1,i)
13
                  ind(i) = out(2,i) ! rank is coerced back to an integer
14
              END DO
15
         END IF
16
17
     Example 5.19
18
         Each process has a non-empty array of values. Find the minimum global value, the
19
     rank of the process that holds it and its index on this process.
20
21
     #define LEN
                      1000
22
23
     float val[LEN];
                             /* local array of values */
24
     int count;
                              /* local number of values */
25
     int myrank, minrank, minindex;
26
     float minval;
27
28
     struct {
29
         float value;
30
         int
                index;
31
     } in, out;
32
33
         /* local minloc */
34
     in.value = val[0];
35
     in.index = 0;
36
     for (i=1; i < count; i++)</pre>
37
          if (in.value > val[i]) {
38
              in.value = val[i];
39
              in.index = i;
40
         }
41
42
          /* global minloc */
43
     MPI_Comm_rank(comm, &myrank);
44
     in.index = myrank*LEN + in.index;
45
     MPI_Reduce( &in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
46
         /* At this point, the answer resides on process root
47
           */
48
```

if	(myrank == root) {
	/* read answer out
	*/
	<pre>minval = out.value;</pre>
	<pre>minrank = out.index / LEN;</pre>
	<pre>minindex = out.index % LEN;</pre>
}	
	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 commute true if commutative; false otherwise.	
OUT op operation (handle)	

```
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 EXTERNAL USER_FN LOGICAL COMMUTE INTEGER OP, IERROR

MPI_OP_CREATE binds a user-defined reduction operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

The argument user_fn is the user-defined function, which must have the following four 47 arguments: invec, inoutvec, len, and datatype. 48

Unofficial Draft for Comment Only

1 The ISO C prototype for the function is the following. 2 typedef void MPI_User_function(void* invec, void* inoutvec, int *len, 3 MPI_Datatype *datatype); 4 The Fortran declarations of the user-defined function user_fn appear below. 5ABSTRACT INTERFACE 6 SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) 7 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 8 TYPE(C_PTR), VALUE :: invec, inoutvec 9 INTEGER :: len 10 TYPE(MPI_Datatype) :: datatype 11 12SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE) 13 <type> INVEC(LEN), INOUTVEC(LEN) 14INTEGER LEN, DATATYPE 15The datatype argument is a handle to the data type that was passed into the call to 16MPI_REDUCE. The user reduce function should be written such that the following holds: 17Let $u[0], \ldots, u[len-1]$ be the len elements in the communication buffer described by the 18 arguments invec, len and datatype when the function is invoked; let $v[0], \ldots, v[len-1]$ be len 19 elements in the communication buffer described by the arguments inoutvec, len and datatype 20when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication 21buffer described by the arguments inoutvec, len and datatype when the function returns; 22 then $w[i] = u[i] \circ v[i]$, for i=0, ..., len-1, where \circ is the reduce operation that the function 23computes. 24Informally, we can think of invec and inoutvec as arrays of len elements that user_fn 25is combining. The result of the reduction over-writes values in inoutvec, hence the name. 26Each invocation of the function results in the pointwise evaluation of the reduce operator 27on len elements: i.e., the function returns in inoutvec[i] the value invec[i] \circ inoutvec[i], for 28 $i=0, \ldots,$ count-1, where \circ is the combining operation computed by the function. 29 30 The len argument allows MPI_REDUCE to avoid calling the function for Rationale. 31each element in the input buffer. Rather, the system can choose to apply the function 32 to chunks of input. In C, it is passed in as a reference for reasons of compatibility 33 with Fortran. 34 By internally comparing the value of the datatype argument to known, global handles, 35it is possible to overload the use of a single user-defined function for several, different 36 data types. (End of rationale.) 37 38 General datatypes may be passed to the user function. However, use of datatypes that 39 are not contiguous is likely to lead to inefficiencies. 40 No MPI communication function may be called inside the user function. MPI_ABORT 41 may be called inside the function in case of an error. 4243 Advice to users. Suppose one defines a library of user-defined reduce functions that 44are overloaded: the datatype argument is used to select the right execution path at each 45invocation, according to the types of the operands. The user-defined reduce function 46cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, 47 the correspondence between the datatype handles and the datatype they represent. 48

This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. Users who plan to mix languages should define their reduction functions accordingly. (End of advice to users.)

Advice to implementation. 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);
                                                                        20
}
                                                                        21
if (rank < groupsize-1) {</pre>
                                                                        22
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
                                                                        23
}
/* answer now resides in process groupsize-1 ... now send to root
 */
if (rank == root) {
                                                                        27
    MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
                                                                        28
}
                                                                        29
if (rank == groupsize-1) {
                                                                        30
    MPI_Send(sendbuf, count, datatype, root, ...);
}
if (rank == root) {
    MPI_Wait(&req, &status);
                                                                        34
}
                                                                        35
```

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.)

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11 12

13

1415

16

17

18

19

 24

25

26

31

32

33

36

37

38

39

40

41

42

43

44

4546

47

```
1
     MPI_OP_FREE(op)
\mathbf{2}
       INOUT
                                             operation (handle)
                 ор
3
4
     int MPI_Op_free(MPI_Op *op)
5
6
     MPI_Op_free(op, ierror)
7
          TYPE(MPI_Op), INTENT(INOUT) :: op
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_OP_FREE(OP, IERROR)
10
          INTEGER OP, IERROR
11
12
         Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
13
14
     Example of User-defined Reduce
15
     It is time for an example of user-defined reduction. The example in this section uses an
16
     intracommunicator.
17
18
     Example 5.20 Compute the product of an array of complex numbers, in C.
19
20
     typedef struct {
21
          double real, imag;
22
     } Complex;
23
24
     /* the user-defined function
25
      */
26
     void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
27
     ſ
28
          int i;
29
          Complex c;
30
          Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
31
32
          for (i=0; i< *len; ++i) {</pre>
33
              c.real = inout->real*in->real -
34
                           inout->imag*in->imag;
35
              c.imag = inout->real*in->imag +
36
                           inout->imag*in->real;
37
              *inout = c;
38
              in++; inout++;
39
          }
40
     }
41
42
     /* and, to call it...
43
      */
44
     . . .
45
46
          /* each process has an array of 100 Complexes
47
           */
48
```

```
Complex a[100], answer[100];
   MPI_Op myOp;
   MPI_Datatype ctype;
   /* explain to MPI how type Complex is defined
     */
   MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
   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);
                                                                                 14
   /* 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.
                                                                                 20
                                                                                 21
 subroutine my_user_function( invec, inoutvec, len, type )
                                                               bind(c)
                                                                                 22
   use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
                                                                                 23
   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
                                                                                 29
       call c_f_pointer(invec, invec_r, (/ len /) )
       call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
       inoutvec_r = invec_r + inoutvec_r
```

All-Reduce 5.9.6

end subroutine

end if

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.

1

 $\mathbf{2}$

3

4

56

7

8

9 10

11

12

13

15

16

17

18 19

 24

25

26

27

28

30

31

32

33

34 35

36

37

38

1 MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm) 2 IN sendbuf starting address of send buffer (choice) 3 OUT recvbuf starting address of receive buffer (choice) 4 5IN number of elements in send buffer (non-negative intecount 6 ger) 7 IN datatype data type of elements of send buffer (handle) 8 IN ор operation (handle) 9 10 IN comm communicator (handle) 11 12int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, 13 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 14MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) 15TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 16TYPE(*), DIMENSION(..) :: recvbuf 17 INTEGER, INTENT(IN) :: count 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19 TYPE(MPI_Op), INTENT(IN) :: op 20TYPE(MPI_Comm), INTENT(IN) :: comm 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 24<type> SENDBUF(*), RECVBUF(*) 25INTEGER COUNT, DATATYPE, OP, COMM, IERROR 26If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as 27MPI_REDUCE except that the result appears in the receive buffer of all the group members. 2829 Advice to implementors. The all-reduce operations can be implemented as a re-30 duce, followed by a broadcast. However, a direct implementation can lead to better 31 performance. (End of advice to implementors.) 32 33 The "in place" option for intracommunicators is specified by passing the value 34 MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is 35 taken at each process from the receive buffer, where it will be replaced by the output data. 36 If comm is an intercommunicator, then the result of the reduction of the data provided 37 by processes in group A is stored at each process in group B, and vice versa. Both groups 38 should provide **count** and **datatype** arguments that specify the same type signature. 39 The following example uses an intracommunicator. 40 41 Example 5.22 42A routine that computes the product of a vector and an array that are distributed 43 across a group of processes and returns the answer at all nodes (see also Example 5.16). 44 4546 47 48

```
1
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
                                                                                            \mathbf{2}
                      ! local slice of array
REAL a(m), b(m,n)
                                                                                            3
REAL c(n)
                        ! result
                                                                                            4
REAL sum(n)
INTEGER n, comm, i, j, ierr
                                                                                            5
                                                                                            6
                                                                                            7
! local sum
                                                                                            8
DO j= 1, n
  sum(j) = 0.0
                                                                                            9
                                                                                            10
  DO i = 1, m
                                                                                            11
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
                                                                                           12
END DO
                                                                                            13
                                                                                           14
                                                                                            15
! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
                                                                                            16
                                                                                            17
                                                                                            18
! return result at all nodes
                                                                                            19
RETURN
END
                                                                                           20
                                                                                           21
                                                                                           22
5.9.7
       Process-Local Reduction
                                                                                           23
The functions in this section are of importance to library implementors who may want to
                                                                                           ^{24}
implement special reduction patterns that are otherwise not easily covered by the standard
                                                                                           25
MPI operations.
                                                                                            26
    The following function applies a reduction operator to local arguments.
                                                                                           27
                                                                                           28
                                                                                           29
MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)
                                                                                           30
  IN
           inbuf
                                        input buffer (choice)
                                                                                           31
                                                                                           32
  INOUT
           inoutbuf
                                        combined input and output buffer (choice)
                                                                                           33
  IN
                                        number of elements in inbuf and inoutbuf buffers (non-
           count
                                                                                           34
                                        negative integer)
                                                                                           35
  IN
           datatype
                                        data type of elements of inbuf and inoutbuf buffers
                                                                                           36
                                        (handle)
                                                                                           37
                                                                                           38
  IN
                                        operation (handle)
            ор
                                                                                           39
                                                                                            40
int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,
                                                                                           41
               MPI_Datatype datatype, MPI_Op op)
                                                                                           42
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
                                                                                           43
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
                                                                                           44
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                            45
    INTEGER, INTENT(IN) :: count
                                                                                            46
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                            47
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                            48
```

1	INTEGER, OPTIONAL, INTENT(C	DUT) :: ierror						
2 3 4 5	MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) <type> INBUF(*), INOUTBUF(*) INTEGER COUNT, DATATYPE, OP, IERROR</type>							
6 7 8 9 10 11 12	and inoutbuf with the result stored e operations in Section 5.9.5. Both i	ion given by op element-wise to the elements of inbuf element-wise in inoutbuf, as explained for user-defined nbuf and inoutbuf (input as well as result) have the count and the same datatype given by datatype. The eried for their commutativity.						
13 14	MPI_OP_COMMUTATIVE(op, comm	ute)						
15	IN op	operation (handle)						
16 17	OUT commute	true if op is commutative, false otherwise (logical) $% \left(\left({{{\left({{\left({\left({\left({\left({\left({\left({\left$						
18 19	<pre>int MPI_Op_commutative(MPI_Op commutative)</pre>	op, int *commute)						
20 21 22 23 24	<pre>MPI_Op_commutative(op, commute, TYPE(MPI_Op), INTENT(IN) :: LOGICAL, INTENT(OUT) :: co INTEGER, OPTIONAL, INTENT(C</pre>	op ommute						
25 26 27 28	MPI_OP_COMMUTATIVE(OP, COMMUTE, LOGICAL COMMUTE INTEGER OP, IERROR	IERROR)						
29 30	5.10 Reduce-Scatter							
31 32 33 34 35 36 37		perations where the result is scattered to all processes atters equal-sized blocks to all processes, while another v in size for each process.						
38								
39 40								
41								
42								
43 44								
45								
46								
47 48								
10								

5.10.1 MPI_REDUCE_SCATTER_BLOCK

MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)	6
οι	JT recvbuf	starting address of receive buffer (choice)	7
IN	recvcount	element count per block (non-negative integer)	8
IN	datatype	data type of elements of send and receive buffers (han-	9
	ddddype	dle)	10
IN	0.7		11
IIN	ор	operation (handle)	12
IN	comm	communicator (handle)	13
			14
int	MPI_Reduce_scatter_block(con	st void* sendbuf, void* recvbuf,	15
	int recvcount, MPI	_Datatype datatype, MPI_Op op,	16
	MPI_Comm comm)		17
			18
MP1_		, recvbuf, recvcount, datatype, op, comm,	19
	ierror)		20
	TYPE(*), DIMENSION(), INTE		21
	TYPE(*), DIMENSION() ::		22
	INTEGER, INTENT(IN) :: recv		23
	TYPE(MPI_Datatype), INTENT(I	v1	24
	TYPE(MPI_Op), INTENT(IN) ::	-	25
	TYPE(MPI_Comm), INTENT(IN) :		26
	INTEGER, OPTIONAL, INTENT(OU	T) :: ierror	27
MPI	REDUCE SCATTER BLOCK (SENDBUF	, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,	28
-	IERROR)	, , , , , ,	29
	<type> SENDBUF(*), RECVBUF(*</type>	:)	30
	INTEGER RECVCOUNT, DATATYPE,		31
	· · · · · · · · · · · · · · · · · · ·		32

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.)

46The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in 47the sendbuf argument on all processes. In this case, the input data is taken from the receive 48 buffer.

Unofficial Draft for Comment Only

5

33

34

35

36

37

38

39

40 41

42

43

44

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.)

12 13 14

9

10

11

5.10.2 MPI_REDUCE_SCATTER

¹⁵ MPI_REDUCE_SCATTER extends the functionality of MPI_REDUCE_SCATTER_BLOCK ¹⁷ such that the scattered blocks can vary in size. Block sizes are determined by the recvcounts ¹⁸ array, such that the i-th block contains recvcounts[i] elements.

19

21

```
<sup>20</sup> MF
```

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

```
IN
                 sendbuf
                                             starting address of send buffer (choice)
22
       OUT
23
                 recvbuf
                                             starting address of receive buffer (choice)
24
       IN
                 recvcounts
                                              non-negative integer array (of length group size) spec-
25
                                              ifying the number of elements of the result distributed
26
                                              to each process.
27
       IN
                 datatype
                                              data type of elements of send and receive buffers (han-
28
                                              dle)
29
30
       IN
                                             operation (handle)
                 ор
31
       IN
                 comm
                                              communicator (handle)
32
33
     int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,
34
                     const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
35
                     MPI_Comm comm)
36
37
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
38
                     ierror)
39
          TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                       sendbuf
40
          TYPE(*), DIMENSION(..) :: recvbuf
41
          INTEGER, INTENT(IN) :: recvcounts(*)
42
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
          TYPE(MPI_Op), INTENT(IN) :: op
44
          TYPE(MPI_Comm), INTENT(IN) :: comm
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
47
                     IERROR)
48
```

<type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of $count = \sum_{i=0}^{n-1} recvcounts[i]$ elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (End of advice to implementors.)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[i] == 0 may not have allocated a receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same recvcounts argument, and provide input vectors of $count = \sum_{i=0}^{n-1} recvcounts[i]$ 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 In	iclusive Scan		35
			36
			37
MPI_SCAN	۱(sendbuf, recvbuf, count, data	type, op, comm)	38
IN	sendbuf	starting address of send buffer (choice)	39
OUT	recvbuf	starting address of receive buffer (choice)	40 41
IN	count	number of elements in input buffer (non-negative in- teger)	42 43
			43 44
IN	datatype	data type of elements of input buffer (handle)	44 45
IN	ор	operation (handle)	45 46
IN	comm	communicator (handle)	46 47
			47 48
			-40

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8

9

10

11

1213

14

15

16

17

18

19

20

21

22

23

 24

25

26

2728

29

30

3132

```
1
     int MPI_Scan(const void* sendbuf, void* recvbuf, int count,
\mathbf{2}
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
3
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
5
          TYPE(*), DIMENSION(..) :: recvbuf
6
          INTEGER, INTENT(IN) :: count
7
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
          TYPE(MPI_Op), INTENT(IN) :: op
9
          TYPE(MPI_Comm), INTENT(IN) :: comm
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
13
          <type> SENDBUF(*), RECVBUF(*)
14
          INTEGER COUNT, DATATYPE, OP, COMM, IERROR
15
         If comm is an intracommunicator, MPI_SCAN is used to perform a prefix reduction on
16
     data distributed across the group. The operation returns, in the receive buffer of the process
17
     with rank i, the reduction of the values in the send buffers of processes with ranks 0, \ldots, j
18
     (inclusive). The routine is called by all group members using the same arguments for count,
19
     datatype, op and comm, except that for user-defined operations, the same rules apply as
20
     for MPI_REDUCE. The type of operations supported, their semantics, and the constraints
21
     on send and receive buffers are as for MPI_REDUCE.
22
          The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in
23
     the sendbuf argument. In this case, the input data is taken from the receive buffer, and
24
     replaced by the output data.
25
         This operation is invalid for intercommunicators.
26
27
     5.11.2 Exclusive Scan
28
29
30
^{31}
     MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)
32
       IN
                 sendbuf
                                             starting address of send buffer (choice)
33
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
34
35
       IN
                 count
                                             number of elements in input buffer (non-negative in-
36
                                             teger)
37
       IN
                 datatype
                                             data type of elements of input buffer (handle)
38
       IN
                                             operation (handle)
                 op
39
40
       IN
                 comm
                                             intracommunicator (handle)
41
42
     int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
43
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
44
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
45
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
46
          TYPE(*), DIMENSION(..) :: recvbuf
47
          INTEGER, INTENT(IN) ::
                                     count
48
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The routine is called by all group members using the same arguments for count, datatype, op and comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE. The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data. The receive buffer on rank 0 is not changed by this operation.

This operation is invalid for intercommunicators.

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan. (*End of rationale.*)

5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.23

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

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

The operator that produces this effect is

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

where

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

Unofficial Draft for Comment Only

 24

```
1
          Note that this is a non-commutative operator. C code that implements it is given
\mathbf{2}
     below.
3
     typedef struct {
4
          double val;
5
          int log;
6
     } SegScanPair;
\overline{7}
8
     /* the user-defined function
9
      */
10
     void segScan(SegScanPair *in, SegScanPair *inout, int *len,
11
                                                             MPI_Datatype *dptr)
12
     {
13
          int i;
14
          SegScanPair c;
15
16
          for (i=0; i< *len; ++i) {</pre>
17
               if (in->log == inout->log)
18
                   c.val = in->val + inout->val;
19
              else
20
                   c.val = inout->val;
21
              c.log = inout->log;
22
              *inout = c;
23
              in++; inout++;
24
          }
25
     }
26
27
          Note that the inout argument to the user-defined function corresponds to the right-
28
     hand operand of the operator. When using this operator, we must be careful to specify that
29
     it is non-commutative, as in the following.
30
^{31}
          int i,base;
32
          SegScanPair a, answer;
33
          MPI_Op
                         myOp;
34
          MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
35
                         disp[2];
          MPI_Aint
36
                         blocklen[2] = { 1, 1};
          int
37
          MPI_Datatype sspair;
38
39
          /* explain to MPI how type SegScanPair is defined
40
           */
41
          MPI_Get_address( &a, disp);
42
          MPI_Get_address( &a.log, disp+1);
43
          base = disp[0];
44
          for (i=0; i<2; ++i) disp[i] -= base;</pre>
45
          MPI_Type_create_struct( 2, blocklen, disp, type, &sspair );
46
          MPI_Type_commit( &sspair );
47
          /* create the segmented-scan user-op
48
           */
```

MPI_Op_create(segScan, 0, &myOp);
...
MPI_Scan(&a, &answer, 1, sspair, myOp, comm);

5.12 Nonblocking Collective Operations

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

The nonblocking collective communication model is similar to the model used for nonblocking point-to-point communication. A nonblocking call initiates a collective operation, 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, i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be safely accessed and modified. Completion does not indicate that other processes have completed or even started the operation (unless otherwise implied by the description of the operation). Completion of a particular nonblocking collective operation also does not indicate completion of any other posted nonblocking collective (or send-receive) operations, whether they are posted before or after the completed operation.

Advice to users. Users should be aware that implementations are allowed, but not required (with exception of MPI_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation. (*End of advice to users.*)

Upon returning from a completion call in which a nonblocking collective operation completes, the MPI_ERROR field in the associated status object is set appropriately, see

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

43

44

45

 $46 \\ 47$

48

1 Section 3.2.5 on page 32. The values of the MPI_SOURCE and MPI_TAG fields are unde- $\mathbf{2}$ fined. It is valid to mix different request types (i.e., any combination of collective requests, 3 I/O requests, generalized requests, or point-to-point requests) in functions that enable mul-4 tiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or $\mathbf{5}$ MPI_CANCEL for a request associated with a nonblocking collective operation. Nonblock-6 ing collective requests created using the APIs described in this section are not persistent. 7However, persistent collective requests can be created using persistent collective operations 8 described in Sections 5.13 and 7.8.

10 11 12

9

13

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.*)

¹⁴ Multiple nonblocking collective operations can be outstanding on a single communi-¹⁵ cator. If the nonblocking call causes some system resource to be exhausted, then it will ¹⁶ fail and generate an MPI exception. Quality implementations of MPI should ensure that ¹⁸ this happens only in pathological cases. That is, an MPI implementation should be able to ¹⁹ support a large number of pending nonblocking operations.

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

- Rationale. Matching blocking and nonblocking collective operations is not allowed
 because the implementation might use different communication algorithms for the two
 cases. Blocking collective operations may be optimized for minimal time to comple tion, while nonblocking collective operations may balance time to completion with
 CPU overhead and asynchronous progression.
- The use of tags for collective operations can prevent certain hardware optimizations. (*End of rationale.*)
- Advice to users. If program semantics require matching blocking and nonblocking collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (End of advice to users.)

In terms of data movement, each nonblocking collective operation has the same effect as
 its blocking counterpart for intracommunicators and intercommunicators after completion.
 Likewise, upon completion, nonblocking collective reduction operations have the same effect
 as their blocking counterparts, and the same restrictions and recommendations on reduction
 orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes. Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

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

5.12.1 Nonblocking Barrier Synchronization

MPI_IBARRIER(comm ,request)

IN	comm	com	municator (handle)			
OUT	request	com	munication request (handle)			
			,			
int MPI_	Ibarrier(MPI_Comm comm, M)	PI_Re	equest *request)			
MPI_Ibar	rier(comm, request, ierro:	r)				
TYPE	C(MPI_Comm), INTENT(IN) ::	com	ım			
TYPE	(MPI_Request), INTENT(OUT)) ::	request			
INTE	GER, OPTIONAL, INTENT(OUT)) ::	ierror			
MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR						

MPI_IBARRIER is a nonblocking version of MPI_BARRIER. By calling MPI_IBARRIER, 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.*)

 $44 \\ 45$

 $\mathbf{2}$

 24

```
202
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.2
             Nonblocking Broadcast
\mathbf{2}
3
4
     MPI_IBCAST(buffer, count, datatype, root, comm, request)
5
       INOUT
                 buffer
                                             starting address of buffer (choice)
6
                                             number of entries in buffer (non-negative integer)
7
       IN
                 count
8
       IN
                 datatype
                                             data type of buffer (handle)
9
       IN
                                             rank of broadcast root (integer)
                  root
10
11
       IN
                 comm
                                             communicator (handle)
12
       OUT
                                             communication request (handle)
                 request
13
14
     int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
15
                    MPI_Comm comm, MPI_Request *request)
16
17
     MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
18
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
19
          INTEGER, INTENT(IN) :: count, root
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
21
          TYPE(MPI_Comm), INTENT(IN) :: comm
22
          TYPE(MPI_Request), INTENT(OUT) :: request
23
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
^{24}
     MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
25
          <type> BUFFER(*)
26
          INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
27
28
          This call starts a nonblocking variant of MPI_BCAST (see Section 5.4).
29
30
     Example using MPI_IBCAST
^{31}
     The example in this section uses an intracommunicator.
32
33
     Example 5.24
34
          Start a broadcast of 100 ints from process 0 to every process in the group, perform some
35
     computation on independent data, and then complete the outstanding broadcast operation.
36
37
          MPI_Comm comm;
38
          int array1[100], array2[100];
39
          int root=0;
40
          MPI_Request req;
41
          . . .
42
          MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
43
          compute(array2, 100);
44
          MPI_Wait(&req, MPI_STATUS_IGNORE);
45
46
47
48
```

5.12.3 Nonblocking Gather

J.12.J	Nonbiocking Gather		2			
MPI_IC	ATHER(sendbuf, sendcour request)	nt, sendtype, recvbuf, recvcount, recvtype, root, comm,	3 4 5			
IN	sendbuf	starting address of send buffer (choice)	6 7			
IN	sendcount	number of elements in send buffer (non-negative integer)	8 9			
IN	sendtype	data type of send buffer elements (handle)	10			
OUT	recvbuf	address of receive buffer (choice, significant only at root)	11 12 13			
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)	14 15			
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	16 17			
IN	root	rank of receiving process (integer)	18 19			
IN	comm	communicator (handle)	20			
OUT	request	communication request (handle)	21			
		- 、 ,	22			
int MP	I_Igather(const void*	<pre>sendbuf, int sendcount, MPI_Datatype sendtype,</pre>	23 24			
		int recvcount, MPI_Datatype recvtype, int root, MPI_Request *request)	25 26			
MPI_Ig	MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 27					
-	root, comm, re	quest, ierror)	28			
		INTENT(IN), ASYNCHRONOUS :: sendbuf	29			
		ASYNCHRONOUS :: recvbuf	30			
		sendcount, recvcount, root	31			
		ENT(IN) :: sendtype, recvtype	32 33			
	PE(MPI_Comm), INTENT() PE(MPI_Request), INTE		33 34			
	TEGER, OPTIONAL, INTE	-	35			
			36			
MPI_IG		UNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	37			
	ROOT, COMM, RE		38			
	<pre>ype> SENDBUF(*), RECVI TECEP SENDCOUNT SEND</pre>	TYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	39			
ΤIV	IEGER SENDCOONI, SEND IERROR	THE, RECOUNT, RECUTTE, ROOT, CORT, REQUEST,	40			
			41			
Tł	is call starts a nonblockin	ng variant of MPI_GATHER (see Section 5.5).	42 43			
			43 44			

1 2	MPI_IGATH	HERV(sendbuf, sendcount, ser comm, request)	ndtype, recvbuf, recvcounts, displs, recvtype, root,				
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)				
5 6	IN	sendcount	number of elements in send buffer (non-negative integer)				
7	IN	sendtype	data type of send buffer elements (handle)				
8 9 10	OUT	recvbuf	address of receive buffer (choice, significant only at root)				
10 11 12 13	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)				
14 15 16 17	IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)				
18 19 20	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)				
21	IN	root	rank of receiving process (integer)				
22	IN	comm	communicator (handle)				
23 24	OUT	request	communication request (handle)				
25 26 27 28 29	<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>						
30 31 32 33 34 35 36 37 38 39 40	 MP1_Igatherv(sendbul, sendcount, sendtype, recvoul, recvounts, dispis, recvtype, root, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendcount, root INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 						
41 42 43 44 45 46 47 48	<pre>MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>						

5.12.4 Nonblocking Scatter

MPI_ISCATTER(sendbuf,	sendcount,	sendtype,	recvbuf,	recvcount,	recvtype,	root,	comm,
request)							

	request)		
IN	sendbuf	address of send buffer (choice, significant only at root)	6 7
IN	sendcount	number of elements sent to each process (non-negative	8
		integer, significant only at root)	9
IN	sendtype	data type of send buffer elements (significant only at	10
		root) (handle)	11
OU	T recvbuf	address of receive buffer (choice)	12 13
IN	recvcount	number of elements in receive buffer (non-negative in-	14
		teger)	15
IN	recvtype	data type of receive buffer elements (handle)	16
IN	root	rank of sending process (integer)	17
IIN	1001	Tank of sending process (integer)	18
IN	comm	communicator (handle)	19
OU	T request	communication request (handle)	20
	·		21
int	MPT Iscatter(const voi	d* sendbuf, int sendcount, MPI_Datatype sendtype,	22
1110		. int recycount. MPI Datatype recytype, int root.	23

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)

MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR

This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).

1 2	MPI_ISCA	ATTERV(sendbuf, sendco comm, request)	unts, displs, sendtype, recvbuf, recvcount, recvtype, root,			
3 4	IN	sendbuf	address of send buffer (choice, significant only at root)			
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	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i			
10 11	IN	sendtype	data type of send buffer elements (handle)			
12	OUT	recvbuf	address of receive buffer (choice)			
$\frac{13}{14}$	IN	recvcount	number of elements in receive buffer (non-negative in-teger)			
15 16	IN	recvtype	data type of receive buffer elements (handle)			
17	IN	root	rank of sending process (integer)			
18	IN	comm	communicator (handle)			
19	OUT	request	communication request (handle)			
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>int MPI_Iscatterv(const void* sendbuf, const int sendcounts[],</pre>					
46 47 48						

5.12.5 Nonblocking Gather-to-all

ΜΡΙ ΙΔΙ	ICATHER(sendbuf send	count, sendtype, recvbuf, recvcount, recvtype, comm,
	request)	count, sendrype, recobal, recocount, recorype, comm,
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
TYP TYP INT TYP TYP INT MPI_IAL (ty) INT	<pre>comm, request, E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT(E(MPI_Request), INTE EGER, OPTIONAL, INTE LGATHER(SENDBUF, SEN COMM, REQUEST, pe> SENDBUF(*), RECVE EGER SENDCOUNT, SEND</pre>	<pre>INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: request NT(OUT) :: ierror DCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, IERROR)</pre>

```
1
     MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm,
\mathbf{2}
                     request)
3
       IN
                  sendbuf
                                              starting address of send buffer (choice)
4
       IN
                  sendcount
                                              number of elements in send buffer (non-negative inte-
5
                                              ger)
6
7
       IN
                  sendtype
                                              data type of send buffer elements (handle)
8
       OUT
                  recvbuf
                                              address of receive buffer (choice)
9
       IN
                                              non-negative integer array (of length group size) con-
                  recvcounts
10
                                              taining the number of elements that are received from
11
                                              each process
12
13
       IN
                  displs
                                              integer array (of length group size). Entry i specifies
14
                                              the displacement (relative to recvbuf) at which to place
15
                                              the incoming data from process i
16
       IN
                                              data type of receive buffer elements (handle)
                  recvtype
17
       IN
                                              communicator (handle)
                  comm
18
19
                                              communication request (handle)
       OUT
                 request
20
21
     int MPI_Iallgatherv(const void* sendbuf, int sendcount,
22
                     MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
23
                     const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
24
                     MPI_Request* request)
25
     MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
26
                     recvtype, comm, request, ierror)
27
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
28
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
29
          INTEGER, INTENT(IN) :: sendcount
30
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
31
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32
          TYPE(MPI_Comm), INTENT(IN) :: comm
33
          TYPE(MPI_Request), INTENT(OUT) ::
34
                                                   request
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
35
36
     MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
37
                     RECVTYPE, COMM, REQUEST, IERROR)
38
          <type> SENDBUF(*), RECVBUF(*)
39
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
40
                      REQUEST, IERROR
41
          This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7).
42
43
44
45
46
47
48
```

5.12.6 Nonblocking All-to-All Scatter/Gather

MPI_IAL	LTOALL(sendbuf, sendco	ount, sendtype, recvbuf, recvcount, recvtype, comm, request)
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 elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
TYPI INTI TYPI TYPI TYPI	E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Comm), INTENT(E(MPI_Request), INTE	sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: request
	EGER, OPTIONAL, INTE	
MPI_IALI	LTOALL (SENDBUF, SEND COMM, REQUEST,	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
<typ< td=""><td><pre>> SENDBUF(*), RECV</pre></td><td></td></typ<>	<pre>> SENDBUF(*), RECV</pre>	
INTE	EGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
This	call starts a nonblockin	ng variant of $MPI_ALLTOALL$ (see Section 5.8).

1 2	MPI_IALL	TOALLV(sendbuf, send. recvtype, comm,	counts, sdispls, sendtype, recvbuf, recvcounts, rdispls, request)
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	IN	sdispls	integer array (of length group size). Entry j specifie the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
0	IN	sendtype	data type of send buffer elements (handle)
1 2	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 specifie 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)
22 23	OUT	request	communication request (handle)
26 27 28		const int rec MPI_Datatype :	<pre>spls[], MPI_Datatype sendtype, void* recvbuf, vcounts[], const int rdispls[], recvtype, MPI_Comm comm, MPI_Request *request)</pre>
29 30 31 32 33 34	TYPE TYPE	<pre>rdispls, recv (*), DIMENSION(), (*), DIMENSION(),</pre>	<pre>ndcounts, sdispls, sendtype, recvbuf, recvcounts, type, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf SYNCHRONOUS :: sendcounts(*), sdispls(*),), rdispls(*)</pre>
85 86		(MPI_Datatype), INT (MPI_Comm), INTENT(TENT(IN) :: sendtype, recvtype (IN) :: comm
37 38		-	ENT(OUT) :: request ENT(OUT) :: ierror
39 40	MPI_IALL		NDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
11 12 13 14	• -	e> SENDBUF(*), REC\ GER SENDCOUNTS(*),	TYPE, COMM, REQUEST, IERROR) /BUF(*) SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), MM, REQUEST, IERROR
45 46 47	This	call starts a nonblocki	ng variant of $MPI_ALLTOALLV$ (see Section 5.8).
48			

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array (of length group size) specifying the num-
		ber of elements to send to each rank (array of non-
		negative integers)
IN	<pre>sendcounts integer array (of length group size) specifying the nu ber of elements to send to each rank (array of n megative integers) sdispls integer array (of length group size). Entry j specifying the nu which to take the outgoing data destined for proce (array of integers) sendtypes array of datatypes (of length group size). Entry j specifying the nu handles) JT recvbuf address of receive buffer (choice) recvcounts integer array (of length group size). Entry i specifying the nu ber of elements that can be received from each r (array of non-negative integers) rdispls integer array (of length group size). Entry i specifying the nu ber of elements that can be received from each r (array of non-negative integers) rdispls integer array (of length group size). Entry i specify integers) recvtypes array of datatypes (of length group size). Entry i specifies the type of data received from process i (array handles) recvtypes array of datatypes (of length group size). Entry i specifies the type of data received from process i (array handles) comm communicator (handle) JT request communicator (handle) MP1_lalltoallw(const void* sendbuf, const int sendcounts[], const int sdispls[], const MP1_Datatype sendtypes[], woid* recvbuf, const int recvcounts[], const int rdispls[], const int sdispls[], const MP1_Comm comm, MP1_Request *request) Jalltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request, ierror) TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf TYPE(MP1_Datatype), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(MP1_Datatype), INTENT(IN), ASYNCHRONOUS :: sendbypes(*), recvtypes(*) TYPE(MP1_Comm), INTENT(IN) :: comm</pre>	
		the displacement in bytes (relative to $sendbuf)$ from
		which to take the outgoing data destined for process j
		(array of integers)
IN	sendtypes	array of datatypes (of length group size). Entry j spec-
0.UT		,
OUT	recvbut	
IN	recvcounts	integer array (of length group size) specifying the num-
	rdicale	
IN	ruispis	
IN	recvtypes	array of datatypes (of length group size). Entry i spec-
		ifies the type of data received from process i (array of
		handles)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
t MPI		
	const int sdispls	[], const MPI_Datatype sendtypes[],
		<pre>onst int recvcounts[], const int rdispls[],</pre>
	const MPI_Datatyp	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm,</pre>
T Tol-	const MPI_Datatyı MPI_Request *requ	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest)</pre>
I_Ial]	const MPI_Datatyr MPI_Request *requ ltoallw(sendbuf, sendco	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, uest) unts, sdispls, sendtypes, recvbuf,</pre>
	const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror)</pre>
TYPI	const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp E(*), DIMENSION(), IN	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: sendbuf</pre>
TYPI TYPI	<pre>const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp E(*), DIMENSION(), IN E(*), DIMENSION(), AS EGER, INTENT(IN), ASYNC</pre>	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: sendbuf YNCHRONOUS :: recvbuf HRONOUS :: sendcounts(*), sdispls(*),</pre>
TYPI TYPI INTI	<pre>const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp E(*), DIMENSION(), IN E(*), DIMENSION(), AS EGER, INTENT(IN), ASYNC recvcounts(*), r</pre>	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: sendbuf YNCHRONOUS :: recvbuf HRONOUS :: sendcounts(*), sdispls(*), displs(*)</pre>
TYPI TYPI INTI	<pre>const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp E(*), DIMENSION(), IN E(*), DIMENSION(), AS EGER, INTENT(IN), ASYNC recvcounts(*), r E(MPI_Datatype), INTENT</pre>	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: sendbuf YNCHRONOUS :: recvbuf HRONOUS :: sendcounts(*), sdispls(*), displs(*)</pre>
TYPI TYPI INTI TYPI	<pre>const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp E(*), DIMENSION(), IN E(*), DIMENSION(), AS EGER, INTENT(IN), ASYNC recvcounts(*), r E(MPI_Datatype), INTENT recvtypes(*)</pre>	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: sendbuf YNCHRONOUS :: recvbuf HRONOUS :: sendcounts(*), sdispls(*), displs(*) (IN), ASYNCHRONOUS :: sendtypes(*),</pre>
TYPI TYPI INTI TYPI TYPI	<pre>const MPI_Datatyp MPI_Request *requ ltoallw(sendbuf, sendco recvcounts, rdisp E(*), DIMENSION(), IN E(*), DIMENSION(), AS EGER, INTENT(IN), ASYNC recvcounts(*), r E(MPI_Datatype), INTENT recvtypes(*)</pre>	<pre>onst int recvcounts[], const int rdispls[], oe recvtypes[], MPI_Comm comm, nest) unts, sdispls, sendtypes, recvbuf, ols, recvtypes, comm, request, ierror) TENT(IN), ASYNCHRONOUS :: sendbuf YNCHRONOUS :: recvbuf HRONOUS :: sendcounts(*), sdispls(*), displs(*) (IN), ASYNCHRONOUS :: sendtypes(*), :: comm</pre>

MPLIALLTOALLW/(conduct condequate edisple conditions recyclute recyclute reliable 1

	212	C	HAPTER 5. COLLECTIVE COMMUNICATION		
1 2 3 4 5	<pre>MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>				
6 7	This call starts a nonblocking variant of $MPI_ALLTOALLW$ (see Section 5.8).				
8 9 10	5.12.7 N	onblocking Reduce			
11 12	MPI_IRED	UCE(sendbuf, recvbuf, count, o	datatype, op, root, comm, request)		
13	IN	sendbuf	address of send buffer (choice)		
14 15 16	OUT	recvbuf	address of receive buffer (choice, significant only at root)		
17 18	IN	count	number of elements in send buffer (non-negative integer)		
19 20	IN	datatype	data type of elements of send buffer (handle)		
20 21	IN	ор	reduce operation (handle)		
22	IN	root	rank of root process (integer)		
23	IN	comm	communicator (handle)		
24 25	OUT	request	communication request (handle)		
26 27 28 29	int MPI_I		uf, void* recvbuf, int count, be, MPI_Op op, int root, MPI_Comm comm, c)		
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<pre>MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,</pre>				
47 48		-	nplementation is explicitly allowed to use different ocking reduction operations that might change the		

order of evaluation of the operations. However, as for MPI_REDUCE, it is strongly recommended that MPI_IREDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processes. (*End of advice to implementors.*)

Advice to users. For operations which are not truly associative, the result delivered upon completion of the nonblocking reduction may not exactly equal the result delivered by the blocking reduction, even when specifying the same arguments in the same order. (*End of advice to users.*)

5.12.8 Nonblocking All-Reduce

15MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request) 16IN sendbuf starting address of send buffer (choice) 17OUT recvbuf 18 starting address of receive buffer (choice) 19 IN count number of elements in send buffer (non-negative inte-20ger) 21IN datatype data type of elements of send buffer (handle) 22 IN operation (handle) 23ор 24 IN comm communicator (handle) 25OUT request communication request (handle) 2627int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count, 28MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 29 MPI_Request *request) 30 31MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request, 32 ierror) 33 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 34 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 35INTEGER, 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, 42IERROR) 43 <type> SENDBUF(*), RECVBUF(*) 44INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 4546This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6). 4748

1 2

3

4

5 6

 $\overline{7}$

8

9

10 11

```
214
```

CHAPTER 5. COLLECTIVE COMMUNICATION

1 5.12.9 Nonblocking Reduce-Scatter with Equal Blocks $\mathbf{2}$ 3 4 MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm, request) 56 IN sendbuf starting address of send buffer (choice) 7 OUT recvbuf 8 starting address of receive buffer (choice) 9 IN recvcount element count per block (non-negative integer) 10 IN data type of elements of send and receive buffers (handatatype 11 dle) 12IN operation (handle) 13 ор 14IN comm communicator (handle) 15OUT communication request (handle) request 1617int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf, 18 int recvcount, MPI_Datatype datatype, MPI_Op op, 19MPI_Comm comm, MPI_Request *request) 2021MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 22 request, ierror) 23TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 24 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 25INTEGER, INTENT(IN) :: recvcount 26TYPE(MPI_Datatype), INTENT(IN) :: datatype 27TYPE(MPI_Op), INTENT(IN) :: op 28TYPE(MPI_Comm), INTENT(IN) :: comm 29TYPE(MPI_Request), INTENT(OUT) :: request 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 31 MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 32 REQUEST, IERROR) 33 34<type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR 35 36 This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-37 tion 5.10.1). 38 39 4041 4243 44 4546 4748

5.12.10 Nonblocking Reduce-Scatter

0.22			2 3				
MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)							
IN	sendbuf	starting address of send buffer (choice)	5				
			6				
Οι	T recvbuf	starting address of receive buffer (choice)	7 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	ор	operation (handle)	13				
IN	comm	communicator (handle)	14 15				
οι	T request	communication request (handle)	15				
			17				
<pre>int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf,</pre>							
<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op,</pre>							
	MPI_Comm comm, MPI_R	equest *request)	20				
MPI_	Ireduce_scatter(sendbuf, recv	ouf, recvcounts, datatype, op, comm,	21 22				
request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*) TYPE(MPI_Datatype), INTENT(IN) :: datatype							
					TYPE(MPI_Op), INTENT(IN) :: (• -	27
					TYPE(MPI_Comm), INTENT(IN) ::	-	28
					TYPE(MPI_Request), INTENT(OUT)		29
	INTEGER, OPTIONAL, INTENT(OUT)) :: ierror	30 31				
MPI	IREDUCE_SCATTER(SENDBUF, RECVI	BUF, RECVCOUNTS, DATATYPE, OP, COMM,	32				
_	REQUEST, IERROR)		33				
	<type> SENDBUF(*), RECVBUF(*)</type>		34				
	INTEGER RECVCOUNTS(*), DATATY	PE, OP, COMM, REQUEST, IERROR	35				
	This call starts a nonblocking variar	at of MPI_REDUCE_SCATTER (see Section $5.10.2$).	36				
	C C		37				
			38 39				
			40				
			41				
			42				
			43				
			44				
			45				
			46				

1

```
216
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.11
              Nonblocking Inclusive Scan
\mathbf{2}
3
4
     MPI_ISCAN(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
                 datatype
                                             data type of elements of input buffer (handle)
11
       IN
                                             operation (handle)
12
                 ор
13
       IN
                 comm
                                             communicator (handle)
14
       OUT
                 request
                                             communication request (handle)
15
16
     int MPI_Iscan(const void* sendbuf, void* recvbuf, int count,
17
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
18
                    MPI_Request *request)
19
20
     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
21
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
22
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
23
          INTEGER, INTENT(IN) :: count
^{24}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          TYPE(MPI_Request), INTENT(OUT) :: request
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
29
     MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
30
          <type> SENDBUF(*), RECVBUF(*)
^{31}
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
32
33
         This call starts a nonblocking variant of MPI_SCAN (see Section 5.11).
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

5.12.12 Nonblocking Exclusive Scan

			4	
			3	
MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)				
IN	sendbuf	starting address of send buffer (choice)	5	
			6	
OUT	recvbuf	starting address of receive buffer (choice)	7 8	
IN	count	number of elements in input buffer (non-negative in- teger)	8 9	
IN	datatype	data type of elements of input buffer (handle)	10	
			11	
IN	ор	operation (handle)	12	
IN	comm	intracommunicator (handle)	13 14	
OUT	request	communication request (handle)	14	
			16	
int MPI_]	exscan(const void* sendb	uf, void* recvbuf, int count,	17	
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,				
	MPI_Request *request	.)	19	
MDT Toxa	an (condbuf recubuf cou	nt, datatype, op, comm, request, ierror)	20	
	(*), DIMENSION(), INTEN		21	
	(*), DIMENSION(), ASYNC	-	22	
	ER, INTENT(IN) :: count		23	
	(MPI_Datatype), INTENT(IN) :: datatvpe	24	
	TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm			
TYPE	MPI_Request), INTENT(OUT)) :: request	27	
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	28	
MDT TEVO	AN CENDRIE DECURIE COIL	NT, DATATYPE, OP, COMM, REQUEST, IERROR)	29	
	> SENDBUF(*), RECVBUF(*)	NI, DAIAIIPE, UP, COMM, REQUESI, IERRUR)	30 31	
• -	ER COUNT, DATATYPE, OP,	COMM REQUEST TERROR	31	
			33	
This o	call starts a nonblocking varia	nt of MPI_EXSCAN (see Section 5.11.2).	34	

Persistent Collective Operations 5.13

Many parallel computation algorithms involve repetitively executing a collective communication operation with the same arguments each time. As with persistent point-to-point operations (see Section 3.9), persistent collective operations allow the MPI programmer to specify operations that will be reused frequently (with fixed arguments). MPI can be designed to select a more efficient way to perform the collective operation based on the parameters specified when the operation is initialized. This "planned-transfer" approach can offer significant performance benefits for programs with repetitive communication patterns.

In terms of data movement, each persistent collective operation has the same effect as 45its blocking and nonblocking counterparts for intracommunicators and intercommunicators 46after completion. Likewise, upon completion, persistent collective reduction operations 47

1 2

217

48

3435

36 37

38

39

40

41

42

43

perform the same operation as their blocking and nonblocking counterparts, and the same
 restrictions and recommendations on reduction orders apply (see also Section 5.9.1).

³ Initialization calls for MPI persistent collective operations are non-local and follow all ⁴ the existing rules for collective operations, in particular ordering; programs that do not ⁵ conform to these restrictions are erroneous. After initialization, all arrays associated with ⁶ input arguments (such as arrays of counts, displacements, and datatypes in the vector ⁷ versions of the collectives) must not be modified until the corresponding persistent request ⁸ is freed with MPI_REQUEST_FREE.

According to the definitions in Section 2.4, the persistent collective initialization pro cedures are incomplete. They are also non-local procedures because they may or may not
 return before they are called in all MPI processes of the process group.

12 13 14

20

21

22

23

24 25

26

27

28

29

30

 31

32

Advice to users. This is one of the exceptions in which incomplete procedures are non-local. (End of advice to users.)

The request argument is an output argument that can be used zero or more times with MPI_START or MPI_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

Rationale. All ordering requirements that an implementation may need to match up collective operations across the communicator are achieved through the ordering requirements of the initialization functions. This enables out-of-order starts for the persistent operations, and particularly supports their use in MPI_STARTALL. (End of rationale.)

Advice to implementors. An MPI implementation should do no worse than duplicating the communicator during the initialization function, caching the input arguments, and calling the appropriate nonblocking collective function, using the cached arguments, during MPI_START. High-quality implementations should be able to amortize setup costs and further optimize by taking advantage of early-binding, such as efficient and effective pre-allocation of certain resources and algorithm selection. (*End* of advice to implementors.)

33 A request must be inactive when it is started. Starting the operation makes the request 34active. Once any process starts a persistent collective operation, it must complete that 35 operation and all other processes in the communicator must eventually start (and complete) 36 the same persistent collective operation. Persistent collective operations cannot be matched 37 with blocking or nonblocking collective operations. Completion of a persistent collective 38 operation makes the corresponding request inactive. After starting a persistent collective 39 operation, all associated send buffers must not be modified and all associated receive buffers 40must not be accessed until the corresponding persistent request is completed.

⁴¹ Completing a persistent collective request, for example using MPI_TEST or
 ⁴² MPI_WAIT, makes it inactive, but does not free the request. This is the same behavior as
 ⁴³ for persistent point-to-point requests. Inactive persistent collective requests can be freed
 ⁴⁴ using MPI_REQUEST_FREE. It is erroneous to free an active persistent collective request.
 ⁴⁵ Persistent collective operations cannot be canceled; it is erroneous to use MPI_CANCEL on
 ⁴⁶ a persistent collective request.

⁴⁷ For every nonblocking collective communication operation in MPI, there is a corre-⁴⁸ sponding persistent collective operation with the analogous API signature.

The collective persistent API signatures include an MPI_INFO object in order to support optimization hints and other information that may be non-standard. Persistent collective operations may be optimized during communicator creation or by the initialization operation of an individual persistent collective. Note that communicator-scoped hints should be provided using MPI_COMM_SET_INFO while, for operation-scoped hints, they are supplied to the persistent collective communication initialization functions using the info argument.

5.13.1 Persistent Barrier Synchronization

			10
MPI BARI	RIER_INIT(comm, info, request	.)	11
IN		,	12
	comm	communicator (handle)	13 14
IN	info	info argument (handle)	14
OUT	request	communication request (handle)	16
			17
int MPI_B	Barrier_init(MPI_Comm com	m, MPI_Info info, MPI_Request *request)	18
MPI_Barri	er_init(comm, info, requ	est, ierror)	19
TYPE	MPI_Comm), INTENT(IN) ::	comm	20
TYPE	<pre>MPI_Info), INTENT(IN) ::</pre>	info	21
	MPI_Request), INTENT(OUT)	-	22
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	23 24
MPI_BARR]	ER_INIT(COMM, INFO, REQU	EST, IERROR)	24
INTEC	ER COMM, INFO, REQUEST, I	IERROR	26
Creat	es a persistent collective com	nunication request for the barrier operation.	27
Oreat	es a persistent conective conn	numeation request for the barrier operation.	28
5132 P	ersistent Broadcast		29
0.10.2			30
			31
MPI_BCAS	ST_INIT(buffer, count, datatyp	e, root, comm, info, request)	32 33
INOUT	buffer	starting address of buffer (choice)	34
IN	count	number of entries in buffer (non-negative integer)	35
		· · · · · · · · · · · · · · · · · · ·	36
IN	datatype	data type of buffer (handle)	37
IN	root	rank of broadcast root (integer)	38
IN	comm	communicator (handle)	39 40
IN	info	info argument (handle)	40
OUT	request	communication request (handle)	42
		- (43
int MPI_H	Scast_init(void* buffer, :	int count, MPI_Datatype datatype,	44
		omm, MPI_Info info, MPI_Request *request)	45
MPT Report	init(huffer count dot	atype, root, comm, info, request, ierror)	46
	(*), DIMENSION(), ASYNCI		47
			48

 $\mathbf{5}$

1 2 3 4 5 6 7 8 9 10 11 12 13	TYPE TYPE TYPE INTEC MPI_BCAST <type INTEC Creat</type 	e> BUFFER(*) GER COUNT, DATATYPE, ROOT	N) :: datatype : comm : info F) :: request
14	5.15.5 F		
15 16 17 18	MPI_GATI	HER_INIT(sendbuf, sendcount info, request)	, sendtype, recvbuf, recvcount, recvtype, root, comm,
19	IN	sendbuf	starting address of send buffer (choice)
20 21	IN	sendcount	number of elements in send buffer (non-negative integer)
22 23	IN	sendtype	data type of send buffer elements (handle)
24 25	OUT	recvbuf	address of receive buffer (choice, significant only at root)
26 27	IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
28 29 30	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
31	IN	root	rank of receiving process (integer)
32	IN	comm	communicator (handle)
33 34	IN	info	info argument (handle)
35	OUT	request	communication request (handle)
36 37 38 39 40 41		MPI_Datatype recvty MPI_Request *reques	pe, void* recvbuf, int recvcount, pe, int root, MPI_Comm comm, MPI_Info info,
42 43 44 45 46 47 48	TYPE TYPE INTE TYPE	root, comm, info, r (*), DIMENSION(), INTEN (*), DIMENSION(), ASYNG GER, INTENT(IN) :: sendo	equest, ierror) NT(IN), ASYNCHRONOUS :: sendbuf CHRONOUS :: recvbuf count, recvcount, root N) :: sendtype, recvtype

TYPE(INTEC MPI_GATHE <type< th=""><th>ROOT, COMM, INFO, RE > SENDBUF(*), RECVBUF(*)</th><th>) :: request) :: ierror T, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,</th><th>1 2 3 4 5 6 7 8 9</th></type<>	ROOT, COMM, INFO, RE > SENDBUF(*), RECVBUF(*)) :: request) :: ierror T, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	1 2 3 4 5 6 7 8 9
Creat	es a persistent collective comr	nunication request for the gather operation.	10 11
MPI_GATI	HERV_INIT(sendbuf, sendcount comm, info, request)	t, sendtype, recvbuf, recvcounts, displs, recvtype, root,	12 13 14
IN	sendbuf	starting address of send buffer (choice)	15
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	16 17 18
IN	sendtype	data type of send buffer elements (handle)	19
OUT	recvbuf	address of receive buffer (choice, significant only at root)	20 21
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)	22 23 24 25
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)	26 27 28 29
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	30 31
IN	root	rank of receiving process (integer)	32 33
IN	comm	communicator (handle)	34
IN	info	info argument (handle)	35
OUT	request	communication request (handle)	36
int MPI_(<pre>const int displs[],</pre>	<pre>sendbuf, int sendcount, e, void* recvbuf, const int recvcounts[], MPI_Datatype recvtype, int root, enfo info, MPI_Request *request)</pre>	37 38 39 40 41 42
TYPE(TYPE) INTEC	recvtype, root, comm (*), DIMENSION(), INTEN (*), DIMENSION(), ASYNC (ER, INTENT(IN) :: sendo		43 44 45 46 47 48

1 2 3 4 5 6	TYPE(TYPE(TYPE(INTEG	<pre>[MPI_Datatype), INTENT(IN [MPI_Comm), INTENT(IN) :: [MPI_Info), INTENT(IN) :: [MPI_Request), INTENT(OUT ER, OPTIONAL, INTENT(OUT</pre>	<pre>comm info) :: request) :: ierror</pre>
7 8 9 10 11	<type< td=""><td>RECVTYPE, ROOT, COMM >> SENDBUF(*), RECVBUF(*)</td><td>NT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, I, INFO, REQUEST, IERROR) RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, , IERROR</td></type<>	RECVTYPE, ROOT, COMM >> SENDBUF(*), RECVBUF(*)	NT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, I, INFO, REQUEST, IERROR) RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, , IERROR
12 13 14 15		es a persistent collective com ersistent Scatter	nunication request for the gathery operation.
16 17 18 19	MPI_SCAT	TER_INIT(sendbuf, sendcount info, request)	, sendtype, recvbuf, recvcount, recvtype, root, comm,
20	IN	sendbuf	address of send buffer (choice, significant only at root)
21 22	IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
23 24 25	IN	sendtype	data type of send buffer elements (significant only at root) (handle)
26	OUT	recvbuf	address of receive buffer (choice)
27 28	IN	recvcount	number of elements in receive buffer (non-negative in-teger)
29 30	IN	recvtype	data type of receive buffer elements (handle)
31	IN	root	rank of sending process (integer)
32	IN	comm	communicator (handle)
33 34	IN	info	info argument (handle)
35	OUT	request	communication request (handle)
36 37 38 39 40 41	int MPI_S		pe, void* recvbuf, int recvcount, pe, int root, MPI_Comm comm, MPI_Info info,
41 42 43 44 45 46 47 48	TYPE(TYPE) INTEC TYPE(recvtype, root, comm	ount, recvcount, root) :: sendtype, recvtype

	E(MPI_Info), INTENT(E(MPI_Request), INTE		1 2
	EGER, OPTIONAL, INTE	-	3
MPT SCA	TTER INIT(SENDBUF, S	ENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	4
		, COMM, INFO, REQUEST, IERROR)	5 6
<ty< td=""><td><pre>pe> SENDBUF(*), RECV</pre></td><td>BUF(*)</td><td>7</td></ty<>	<pre>pe> SENDBUF(*), RECV</pre>	BUF(*)	7
INT	EGER SENDCOUNT, SEND REQUEST, IERR	TYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, OR	8 9
Crea	ates a persistent collection	ve communication request for the scatter operation.	10 11
MPI_SC	ATTERV_INIT(sendbuf, root, comm, info,	sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, request)	12 13 14
IN	sendbuf	address of send buffer (choice, significant only at root)	15
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	16 17 18
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	19 20 21
IN	sendtype	data type of send buffer elements (handle)	22
OUT	recvbuf	address of receive buffer (choice)	23 24
IN	recvcount	number of elements in receive buffer (non-negative in-teger)	24 25 26
IN	recvtype	data type of receive buffer elements (handle)	27
IN	root	rank of sending process (integer)	28 29
IN	comm	communicator (handle)	30
IN	info	info argument (handle)	31
OUT	request	communication request (handle)	32
			33 34
int MPI		<pre>void* sendbuf, const int sendcounts[],</pre>	35
	-	<pre>ls[], MPI_Datatype sendtype, void* recvbuf,</pre>	36
		<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>	37
WDT O			38 39
MP1_Sca		sendcounts, displs, sendtype, recvbuf, vtype, root, comm, info, request, ierror)	40
TYP		INTENT(IN), ASYNCHRONOUS :: sendbuf	41
		ASYNCHRONOUS :: recvbuf	42
		YNCHRONOUS :: sendcounts(*), displs(*)	43
	EGER, INTENT(IN) ::		44 45
	E(MPI_Datatype), INT E(MPI_Comm), INTENT(ENT(IN) :: sendtype, recvtype	45 46
	E(MPI_COMM), INTENI(E(MPI_Info), INTENI(47
	E(MPI_Request), INTE		48

1	INTEC	GER, OPTIONAL, INTENT(OUT)) :: ierror
2 3 4 5 6 7	<type< th=""><th>RECVCOUNT, RECVTYPE, > SENDBUF(*), RECVBUF(*)</th><th>UNTS, DISPLS, SENDTYPE, RECVBUF, ROOT, COMM, INFO, REQUEST, IERROR) (*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, , IERROR</th></type<>	RECVCOUNT, RECVTYPE, > SENDBUF(*), RECVBUF(*)	UNTS, DISPLS, SENDTYPE, RECVBUF, ROOT, COMM, INFO, REQUEST, IERROR) (*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, , IERROR
8 9	Creat	es a persistent collective com	nunication request for the scattery operation.
10 11 12	5.13.5 P	ersistent Gather-to-all	
12 13 14 15	MPI_ALLC	GATHER_INIT(sendbuf, sendco info, request)	unt, sendtype, recvbuf, recvcount, recvtype, comm,
16	IN	sendbuf	starting address of send buffer (choice)
17 18	IN	sendcount	number of elements in send buffer (non-negative integer)
19 20	IN	sendtype	data type of send buffer elements (handle)
21	OUT	recvbuf	address of receive buffer (choice)
22 23	IN	recvcount	number of elements received from any process (non-negative integer)
24 25	IN	recvtype	data type of receive buffer elements (handle)
26	IN	comm	communicator (handle)
27	IN	info	info argument (handle)
28 29	OUT	request	communication request (handle)
30 31 32 33 34	int MPI_4	MPI_Datatype sendtyp	* sendbuf, int sendcount, be, void* recvbuf, int recvcount, be, MPI_Comm comm, MPI_Info info,
35 36 37 38	TYPE	recvtype, comm, info	T(IN), ASYNCHRONOUS :: sendbuf
39		GER, INTENT(IN) :: sendco	
40	TYPE	(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41 42		(MPI_Comm), INTENT(IN) ::	
43		(MPI_Info), INTENT(IN) :: (MPI_Request), INTENT(OUT)	
44		GER, OPTIONAL, INTENT(OUT)	-
45			OUNT, SENDTYPE, RECVBUF, RECVCOUNT,
46 47		RECVTYPE, COMM, INFO	
48	<type< td=""><td><pre>> SENDBUF(*), RECVBUF(*)</pre></td><td></td></type<>	<pre>> SENDBUF(*), RECVBUF(*)</pre>	

	INTEGER SENDCOUNT, IERROR	SENDTYPE,	RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	$\frac{1}{2}$	
(Creates a persistent c	ollective com	munication request for the allgather operation.	3 4	
MPI_	ALLGATHERV_INIT(: comm, infc		count, sendtype, recvbuf, recvcounts, displs, recvtype,	5 6 7	
IN	sendbuf		starting address of send buffer (choice)	8	
IN	sendcount		number of elements in send buffer (non-negative inte-	9 10	
IIN	Schucount		ger)	11	
IN	sendtype		data type of send buffer elements (handle)	12	
OU			address of receive buffer (choice)	13	
IN	recvcounts		non-negative integer array (of length group size) con-	14 15	
IIN	recvcounts		taining the number of elements that are received from each process	16 17	
IN	displs		integer array (of length group size). Entry i specifies	18	
	·		the displacement (relative to recvbuf) at which to place	19	
			the incoming data from process i	20 21	
IN	recvtype		data type of receive buffer elements (handle)	21	
IN	comm		communicator (handle)	23	
IN	info		info argument (handle)	24	
OU	T request		communication request (handle)	25	
				26 27	
int 1	<pre>IPI_Allgatherv_ini</pre>	t(const voi	.d* sendbuf, int sendcount,	28	
			pe, void* recvbuf, const int recvcounts[],	29	
		-	MPI_Datatype recvtype, MPI_Comm comm,	30	
	MP1_INIO	inio, MPI_I	Request* request)	31 32	
MPI_	-		count, sendtype, recvbuf, recvcounts,	33	
	-		omm, info, request, ierror) NT(IN), ASYNCHRONOUS :: sendbuf	34	
	TYPE(*), DIMENSION			35	
	INTEGER, INTENT(IN			36	
			<pre>NOUS :: recvcounts(*), displs(*)</pre>	37 38	
	• -		I) :: sendtype, recvtype	39	
	TYPE(MPI_Comm), IN			40	
	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request				
	INTEGER, OPTIONAL,		-	42	
мрт	ALLGATHERV INIT(SF	NDBUE SENF	COUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	$\frac{43}{44}$	
··· ±_/			DMM, INFO, REQUEST, IERROR)	45	
	<pre>type> SENDBUF(*),</pre>	-		46	
			RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	47	
INFO, REQUEST, IERROR 48					

```
226
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
          Creates a persistent collective communication request for the allgathery operation.
\mathbf{2}
3
     5.13.6 Persistent All-to-All Scatter/Gather
4
5
6
     MPI_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info,
7
                    request)
8
                 sendbuf
       IN
                                             starting address of send buffer (choice)
9
10
       IN
                 sendcount
                                             number of elements sent to each process (non-negative
11
                                             integer)
12
       IN
                 sendtype
                                             data type of send buffer elements (handle)
13
       OUT
                 recvbuf
                                             address of receive buffer (choice)
14
15
       IN
                 recvcount
                                             number of elements received from any process (non-
16
                                             negative integer)
17
       IN
                                             data type of receive buffer elements (handle)
                 recvtype
18
       IN
                 comm
                                             communicator (handle)
19
20
       IN
                 info
                                             info argument (handle)
21
       OUT
                                             communication request (handle)
                 request
22
23
     int MPI_Alltoall_init(const void* sendbuf, int sendcount,
24
                    MPI_Datatype sendtype, void* recvbuf, int recvcount,
25
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
26
                    MPI_Request *request)
27
28
     MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
29
                    recvtype, comm, info, request, ierror)
30
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
^{31}
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
32
          INTEGER, INTENT(IN) :: sendcount, recvcount
33
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
          TYPE(MPI_Comm), INTENT(IN) :: comm
35
          TYPE(MPI_Info), INTENT(IN) ::
                                             info
36
          TYPE(MPI_Request), INTENT(OUT) :: request
37
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
39
                    RECVTYPE, COMM, INFO, REQUEST, IERROR)
40
          <type> SENDBUF(*), RECVBUF(*)
41
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
42
                     IERROR
43
44
          Creates a persistent collective communication request for the alltoall operation.
45
46
47
48
```

MPI_ALLT	OALLV_INIT(sendbuf, sendcou recvtype, comm, info, req	nts, sdispls, sendtype, recvbuf, recvcounts, rdispls, uest)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	4 5 6
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10
OUT	recvbuf	address of receive buffer (choice)	11 12
IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank	13 14 15
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	16 17 18 19
IN	recvtype	data type of receive buffer elements (handle)	20
IN	comm	communicator (handle)	21
IN	info	info argument (handle)	22 23
OUT	request	communication request (handle)	23 24
			25
int MPI_A		sendbuf, const int sendcounts[],	26
	-	<pre>MPI_Datatype sendtype, void* recvbuf, [], const int rdispls[],</pre>	27 28
		e, MPI_Comm comm, MPI_info info,	29
	MPI_Request *request))	30
MPI_Allto	allv_init(sendbuf, sendco	ounts, sdispls, sendtype, recvbuf,	31
	recvcounts, rdispls,	recvtype, comm, info, request, ierror)	32 33
		(IN), ASYNCHRONOUS :: sendbuf	34
	*), DIMENSION(), ASYNCH		35
INIEG	recvcounts(*), rdisp	<pre>IOUS :: sendcounts(*), sdispls(*), ls(*)</pre>	36
TYPE(MPI_Datatype), INTENT(IN)		37
	MPI_Comm), INTENT(IN) ::		38 39
	<pre>MPI_Info), INTENT(IN) ::</pre>		40
	MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	-	41
			42
MPI_ALLTO		NUNTS, SDISPLS, SENDTYPE, RECVBUF,	43 44
<tvpe< td=""><td><pre>> SENDBUF(*), RECVBUF(*)</pre></td><td>RECVTYPE, COMM, INFO, REQUEST, IERROR)</td><td>45</td></tvpe<>	<pre>> SENDBUF(*), RECVBUF(*)</pre>	RECVTYPE, COMM, INFO, REQUEST, IERROR)	45
• 1		<pre>s(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	46
	RECVTYPE, COMM, INFO	, REQUEST, IERROR	47
			48

	228	CH	IAPTER 5. COLLECTIVE COMMUNICATION
1 2	Create	es a persistent collective comm	nunication request for the alltoally operation.
3 4 5	MPI_ALLT	OALLW_INIT(sendbuf, sendcou recvtypes, comm, info, rec	unts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, quest)
6	IN	sendbuf	starting address of send buffer (choice)
7 8 9 10	IN	sendcounts	integer array (of length group size) specifying the num- ber of elements to send to each rank (array of non- negative integers)
11 12 13 14	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)
15 16 17	IN	sendtypes	array of datatypes (of length group size). Entry j spec- ifies the type of data to send to process j (array of handles)
18 19	OUT	recvbuf	address of receive buffer (choice)
20 21 22	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)
23 24 25 26	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)
27 28 29 30	IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)
30 31	IN	comm	communicator (handle)
32	IN	info	info argument (handle)
33 34	OUT	request	communication request (handle)
35 36 37 38 39 40	int MPI_A	<pre>const int sdispls[], void* recvbuf, const</pre>	<pre>sendbuf, const int sendcounts[], const MPI_Datatype sendtypes[], int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm, MPI_Info info,</pre>
41 42 43 44 45 46 47 48	TYPE(TYPE(INTEG	recvcounts, rdispls, *), DIMENSION(), INTENT *), DIMENSION(), ASYNCH ER, INTENT(IN), ASYNCHRON recvcounts(*), rdisp	<pre>IOUS :: sendcounts(*), sdispls(*),</pre>

TYPE	(MPI_Comm), INTENT(IN) : (MPI_Info), INTENT(IN) : (MDI_Document)INTENT(OU	: info	1 2 3
	(MPI_Request), INTENT(OU GER, OPTIONAL, INTENT(OU	1	4
MPI_ALLT	DALLW_INIT(SENDBUF, SEND RECVCOUNTS, RDISPLS e> SENDBUF(*), RECVBUF(* GER SENDCOUNTS(*), SDISP	COUNTS, SDISPLS, SENDTYPES, RECVBUF, 5, RECVTYPES, COMM, INFO, REQUEST, IERROR)	5 6 7 8 9 10
Creat	es a persistent collective con	nmunication request for the alltoallw operation.	11 12
5.13.7 P	Persistent Reduce		12 13 14 15
MPI_RED	UCE_INIT(sendbuf, recvbuf, o	count, datatype, op, root, comm, info, request)	16 17
IN	sendbuf	address of send buffer (choice)	18
OUT	recvbuf	address of receive buffer (choice, significant only at root)	19 20
IN	count	number of elements in send buffer (non-negative integer)	21 22 23
IN	datatype	data type of elements of send buffer (handle)	24
IN	ор	reduce operation (handle)	25
IN	root	rank of root process (integer)	26 27
IN	comm	communicator (handle)	28
IN	info	info argument (handle)	29
OUT	request	communication request (handle)	30 31
int MPT 1	Reduce init(const void*	sendbuf, void* recvbuf, int count,	32
		<pre>/pe, MPI_Op op, int root, MPI_Comm comm,</pre>	33 34
	MPI_Info info, MPI_	Request *request)	35
MPI_Redu	ce_init(sendbuf, recvbuf	, count, datatype, op, root, comm, info,	36
	request, ierror)		37
		NT(IN), ASYNCHRONOUS :: sendbuf	38 39
	(*), DIMENSION(), ASYN GER, INTENT(IN) :: coun		40
	(MPI_Datatype), INTENT(I		41
	(MPI_Op), INTENT(IN) ::	V1	42
TYPE	(MPI_Comm), INTENT(IN) :	: comm	43
	(MPI_Info), INTENT(IN) :		44 45
	(MPI_Request), INTENT(OU	-	45 46
TN.LE(GER, OPTIONAL, INTENT(OU	1) :: lerror	47
			48

CHAPTER 5. COLLECTIVE COMMUNICATION

Creates a persistent collective communication request for the reduce operation. 5.13.8 Persistent All-Reduce MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request) IN sendbuf starting address of send buffer (choice) OUT recvbuf starting address of receive buffer (choice) IN count number of elements in send buffer (non-negative inte- ger) IN datatype data type of elements of send buffer (handle) IN op operation (handle) IN comm communicator (handle) IN info info argument (handle) OUT request communication request (handle) IN info info argument (handle) OUT request communication request (handle) IN mPI_Lallreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Latatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: comm TYPE(MPI_Domm, INTENT(IN) :: comm TYPE(MPI_Domm, INTENT(IN) :: request INTEGER, INTENT(IN) :: comm TYPE(MPI_Do), INTENT(IN) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, TERROR) <tp> <tp> Kupe> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR</tp></tp>	1 2 3 4	MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR</type>			
<pre>5.13.8 Persistent All-Reduce MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request) N sendbuf starting address of send buffer (choice) OUT recvbuf starting address of receive buffer (choice) N count number of elements in send buffer (non-negative inte- ger) N datatype data type of elements of send buffer (handle) N op operation (handle) N comm communicator (handle) N info info argument (handle) OUT request communication request (handle) int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN) :: comt TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf TYPE(MPI_Comm), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Request), INTENT(IN) :: ierror MPI_ALLREDUCE_INIT(SENDEUF, RECVEUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR <type> SENDBUF(*), RECVEUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR Creates a persistent collective communication request for the allreduce operation.</type></pre>		Creat	tes a persistent collective com	nunication request for the reduce operation.	
MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request) IN sendbuf starting address of send buffer (choice) IN count number of elements in send buffer (non-negative inte- ger) IN datatype data type of elements of send buffer (non-negative inte- ger) IN ount number of elements of send buffer (nandle) IN op operation (handle) IN comm communicator (handle) IN info info argument (handle) OUT request communication request (handle) IN mfi_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: com TYPE(MPI_Dop), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(IN) :: ierror MPI_ALLREDUCE_INTI(SENDEUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR <type> SENDEUF(*), IERROR <tp>Creates a persistent collective communication request for the allreduce operation. <th>8 9</th><th>5.13.8 P</th><th>Persistent All-Reduce</th><th></th></tp></type>	8 9	5.13.8 P	Persistent All-Reduce		
IN sendbuf starting address of send buffer (choice) OUT recvbuf starting address of receive buffer (choice) IN count number of elements in send buffer (non-negative inte- ger) IN datatype data type of elements of send buffer (handle) IN op operation (handle) IN op operation (handle) IN op operation (handle) IN info info argument (handle) OUT request communicator (handle) IN info info argument (handle) OUT request communication request (handle) IN mfi_antype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Info info, MPI_Request *request) TYPE(*), DIMENSION(), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: com TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: op TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(IN) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) </th <th></th> <th>MPI ALLE</th> <th>REDUCE INIT(sendbuf_recybu</th> <th>f count datatype on comm info request)</th>		MPI ALLE	REDUCE INIT(sendbuf_recybu	f count datatype on comm info request)	
OUT recvbuf starting address of receive buffer (choice) IN count number of elements in send buffer (non-negative inte- ger) IN datatype data type of elements of send buffer (handle) IN op operation (handle) IN comm communicator (handle) IN info info argument (handle) OUT request communication request (handle) OUT request communication request (handle) IN mfi_Lallreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) request; ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: op TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Info), INTENT(IN) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) REQUEST, IERROR) 'type' SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR			Ύ		
15 IN count number of elements in send buffer (non-negative inte- ger) 17 IN datatype data type of elements of send buffer (handle) 19 IN op operation (handle) 19 IN op operation (handle) 19 IN comm communicator (handle) 20 IN comm communication request (handle) 21 IN info info argument (handle) 22 OUT request communication request (handle) 23 OUT request communication request (handle) 24 int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, 25 int MPI_allreduce_init(const void* sendbuf, void* recvbuf, int count, 26 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 27 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, 28 request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf 29 TYPE(MPI_Datatype), INTENT(IN) :: datatype 20 TYPE(MPI_Op), INTENT(IN) :: op 30 TYPE(MPI_Icho), INTENT(IN) :: info 31 TYPE(MPI_Lequest), I					
17 IN datatype data type of elements of send buffer (handle) 19 IN op operation (handle) 20 IN comm communicator (handle) 21 IN info info argument (handle) 22 OUT request communication request (handle) 23 OUT request communication request (handle) 24 int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, 26 int MPI_Lallreduce_init(const void* sendbuf, void* recvbuf, int count, 27 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 28 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, 29 request, ierror 30 TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf 31 TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf 32 INTEGER, INTENT(IN) :: count 33 TYPE(MPI_Op), INTENT(IN) :: op 34 TYPE(MPI_Op), INTENT(IN) :: info 35 TYPE(MPI_Request), INTENT(OUT) :: request 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,	15			number of elements in send buffer (non-negative inte-	
 IN op operation (handle) IN comm communicator (handle) IN info info argument (handle) OUT request communication request (handle) int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) <th>17</th><th>IN</th><th>datatype</th><th>- ,</th>	17	IN	datatype	- ,	
20 IN comm communicator (handle) 21 IN info info argument (handle) 23 OUT request communication request (handle) 24 int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) 26 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) 27 TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf 28 TYPE(MPI_Datatype), INTENT(IN) ASYNCHRONOUS :: sendbuf 29 TYPE(MPI_Datatype), INTENT(IN) :: count 30 TYPE(MPI_Datatype), INTENT(IN) :: datatype 31 TYPE(MPI_Datatype), INTENT(IN) :: comm 32 TYPE(MPI_Op), INTENT(IN) :: comm 33 TYPE(MPI_Comm), INTENT(IN) :: info 34 TYPE(MPI_Request), INTENT(OUT) :: request 35 TYPE(MPI_Request), INTENT(OUT) :: ierror 36 MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, 35 REQUEST, IERROR 36 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 37 TYPE 38 INTEGER collective communication request for the allreduce operation.		IN	ор	operation (handle)	
IN INTO into argument (handle) OUT request communication request (handle) int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR</type>		IN	comm	communicator (handle)	
23 OUT request communication request (handle) 24 int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, 26 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 27 MPI_Info info, MPI_Request *request) 28 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, 29 request, ierror) 30 TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf 31 TYPE(*), DIMENSION(), ASYNCHRONOUS :: 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) :: info 36 TYPE(MPI_Info), INTENT(IN) :: info 37 TYPE(MPI_Request), INTENT(OUT) :: request 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, 38 REQUEST, IERROR) 39 <type> SENDBUF(*), RECVBUF(*) 31 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 34 Creates a persistent collective communication request for the allreduce operation.</type>		IN	info	info argument (handle)	
<pre>int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: ierquest INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) </pre>		OUT	request	communication request (handle)	
46 47	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request) MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVEUF(*)</type> INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR Creates a persistent collective communication request for the allreduce operation. 			

5.13. PI	ERSISTENT COLLECT	TIVE OPERATIONS 231
5.13.9	Persistent Reduce-Scat	ter with Equal Blocks
MPI_RE	DUCE_SCATTER_BLOC info, request)	K_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm,
IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	element count per block (non-negative integer)
IN	datatype	data type of elements of send and receive buffers (han- dle)
IN	ор	operation (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)
		MPI_Datatype datatype, MPI_Op op, MPI_Info info, MPI_Request *request)
TYPI TYPI INTI TYPI TYPI TYPI TYPI	<pre>comm, info, re E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) ::</pre>	ENT(IN) :: datatype) :: op IN) :: comm IN) :: info NT(OUT) :: request
<tyj INTI</tyj 	COMM, INFO, RE pe> SENDBUF(*), RECV EGER RECVCOUNT, DATA	BUF(*) TYPE, OP, COMM, INFO, REQUEST, IERROR
blocks op	*	re communication request for the reduce-scatter with equal

```
232
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
     5.13.10 Persistent Reduce-Scatter
1
\mathbf{2}
3
4
     MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info, re-
5
                    quest)
6
       IN
                 sendbuf
                                             starting address of send buffer (choice)
7
       OUT
8
                 recvbuf
                                             starting address of receive buffer (choice)
9
       IN
                 recvcounts
                                             non-negative integer array specifying the number of
10
                                             elements in result distributed to each process. Array
11
                                             must be identical on all calling processes.
12
       IN
                 datatype
                                             data type of elements of input buffer (handle)
13
14
       IN
                 op
                                             operation (handle)
15
       IN
                 comm
                                             communicator (handle)
16
       IN
                 info
                                             info argument (handle)
17
       OUT
18
                 request
                                             communication request (handle)
19
20
     int MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf,
21
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
22
                    MPI_Comm comm, MPI_Info info, MPI_Request *request)
23
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
24
                    info, request, ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                      sendbuf
26
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Op), INTENT(IN) :: op
30
          TYPE(MPI_Comm), INTENT(IN) :: comm
^{31}
          TYPE(MPI_Info), INTENT(IN) ::
                                             info
32
          TYPE(MPI_Request), INTENT(OUT) :: request
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
36
                    INFO, REQUEST, IERROR)
37
          <type> SENDBUF(*), RECVBUF(*)
38
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR
39
          Creates a persistent collective communication request for the reduce-scatter operation.
40
41
42
43
44
45
46
47
48
```

5.13. PERSISTENT COLLECTIVE OPERATIONS

5.13.11 Persistent Inclusive Scan

5.15	II Persistent inclusive Scan		2
MPI	SCAN INIT(sendbuf recybuf	count, datatype, op, comm, info, request)	3 4
IN	sendbuf	starting address of send buffer (choice)	5
			6
OL		starting address of receive buffer (choice)	7 8
IN	count	number of elements in input buffer (non-negative in- teger)	9 10
IN	datatype	data type of elements of input buffer (handle)	11
IN	ор	operation (handle)	12
IN	comm	communicator (handle)	13
IN	info	info argument (handle)	14
οι		communication request (handle)	15 16
00		communication request (nancie)	17
int	MPI Scan init(const void*	<pre>sendbuf, void* recvbuf, int count,</pre>	18
		atype, MPI_Op op, MPI_Comm comm,	19
	MPI_Info info, M	PI_Request *request)	20
мрт	Scan init(sendbuf recybu	f, count, datatype, op, comm, info, request,	21
···	ierror)	1, 00410, 44040, po, op, 001111, 1110, 1044000,	22 23
	TYPE(*), DIMENSION(), I	NTENT(IN), ASYNCHRONOUS :: sendbuf	24
	TYPE(*), DIMENSION(), A		25
	INTEGER, INTENT(IN) :: c		26
	TYPE(MPI_Datatype), INTEN		27
	TYPE(MPI_Op), INTENT(IN) TYPE(MPI_Comm), INTENT(IN	-	28
	TYPE(MPI_COMM), INTENT(IN TYPE(MPI_Info), INTENT(IN		29
	TYPE(MPI_Request), INTENT		30 31
	INTEGER, OPTIONAL, INTENT	-	32
мрт	SCAN INIT (SENDBUE BECVBU	F, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,	33
··· +_	IERROR)		34
	<type> SENDBUF(*), RECVBU</type>	F(*)	35
	INTEGER COUNT, DATATYPE,	OP, COMM, INFO, REQUEST, IERROR	36
	Creates a persistent collective	communication request for the inclusive scan operation.	37 38
	ereates a persistent concerne		39
			40
			41
			42
			43
			44

```
1
     5.13.12 Persistent Exclusive Scan
\mathbf{2}
3
4
     MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, 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
                 ор
13
       IN
                 comm
                                              intracommunicator (handle)
14
       IN
                 info
                                              info argument (handle)
15
16
       OUT
                 request
                                              communication request (handle)
17
18
     int MPI_Exscan_init(const void* sendbuf, void* recvbuf, int count,
19
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
20
                     MPI_Info info, MPI_Request *request)
21
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
22
                     ierror)
23
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN) :: count
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Info), INTENT(IN) ::
                                              info
30
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
32
33
     MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
34
                     IERROR)
35
          <type> SENDBUF(*), RECVBUF(*)
36
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
37
          Creates a persistent collective communication request for the exclusive scan operation.
38
39
40
     5.14
             Correctness
41
42
     A correct, portable program must invoke collective communications so that deadlock will not
43
     occur, whether collective communications are synchronizing or not. The following examples
44
     illustrate dangerous use of collective routines on intracommunicators.
45
     Example 5.25
46
          The following is erroneous.
47
48
```

CHAPTER 5. COLLECTIVE COMMUNICATION

<pre>switch(rank) {</pre>	1
case 0:	2
<pre>MPI_Bcast(buf1, count, type, 0, comm);</pre>	3
<pre>MPI_Bcast(buf2, count, type, 1, comm);</pre>	4
break;	5
case 1:	6
<pre>MPI_Bcast(buf2, count, type, 1, comm);</pre>	7
<pre>MPI_Bcast(buf1, count, type, 0, comm);</pre>	8
break;	9
}	10
	11

We assume that the group of comm is $\{0,1\}$. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

Example 5.26

The following is erroneous.

```
switch(rank) {
   case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
   case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
   case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
}
```

}

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

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

Example 5.27

The following is erroneous.

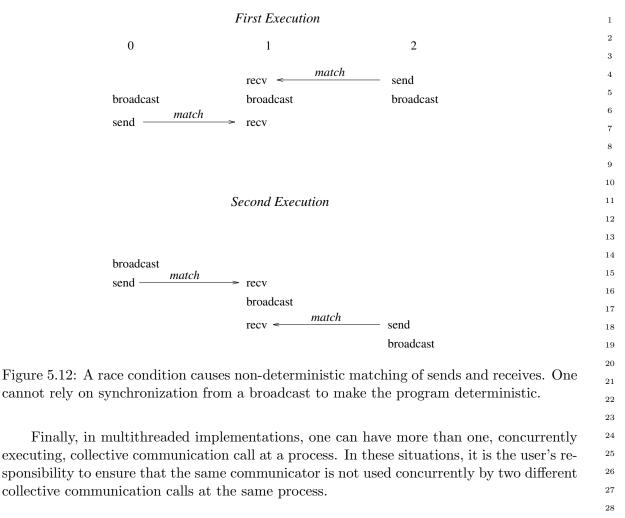
 24

 31

 $41 \\ 42$

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

example illustrates the fact that one should not rely on collective communication functions
 to have particular synchronization effects. A program that works correctly only when the
 first execution occurs (only when broadcast is synchronizing) is erroneous.



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

29

30 31

32

33

34 35

36

37

38

39

40

41

42

43 44 45

46

47

```
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.

```
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:
            MPI_Ibarrier(comm, &reqs[0]);
9
            MPI_Send(buf, count, dtype, 1, tag, comm);
10
            MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
11
            break;
12
          case 1:
13
            MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
14
            MPI_Ibarrier(comm, &reqs[1]);
15
16
            MPI_Waitall(2, regs, MPI_STATUSES_IGNORE);
            break;
17
     }
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
```

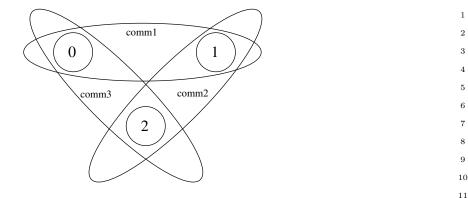


Figure 5.13: Example with overlapping communicators.

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 [55] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 Features Needed to Support Libraries

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

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

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

 24

6.1.2 MPI's Support for Libraries

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

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

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

¹⁹ Caching. Communicators (see below) provide a "caching" mechanism that allows one to ²⁰ associate new attributes with communicators, on par with MPI built-in features. This can ²¹ be used by advanced users to adorn communicators further, and by MPI to implement ²² some communicator functions. For example, the virtual-topology functions described in ²³ Chapter 7 are likely to be supported this way.

24

1

2

3

4 5

6

7 8

9 10

11

12

18

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

31

36

37

38

39

40

41

42

43

44

45 46

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

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

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

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

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

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

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

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

Unofficial Draft for Comment Only

 24

 31

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

6.2 Basic Concepts

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

6.2.1 Groups

1

 $\mathbf{2}$

3 4

5 6

> 7 8

9

20 21

22

23 24

25

26

27

28

34

35

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

There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no
 members. The predefined constant MPI_GROUP_NULL is the value used for invalid group
 handles.

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

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

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

6.2.2 Contexts

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

 $41 \\ 42$

43

44

45

46

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

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

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

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

6.2.3 Intra-Communicators

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

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

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

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

6.2.4 Predefined Intra-Communicators

An initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

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

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

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28 29

30

31

32 33

34

35

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)
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)
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)				
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_	<pre>int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[], MPI_Group group2, int ranks2[])</pre>			
MPI_Grou	p_translate_ranks(group1,	n, ranks1, group2, ranks2, ierror)	13	
	(MPI_Group), INTENT(IN) :		14 15	
	GER, INTENT(IN) :: n, ra GER, INTENT(OUT) :: rank		16	
	GER, OPTIONAL, INTENT(OUT		17	
		N, RANKS1, GROUP2, RANKS2, IERROR)	18 19	
		GROUP2, RANKS2(*), IERROR	20	
This	function is important for deter	mining the relative numbering of the same processes	21	
	-	one knows the ranks of certain processes in the group	22 23	
	, , , , , , , , , , , , , , , , , , ,	to know their ranks in a subset of that group.	23 24	
		input to MPI_GROUP_TRANSLATE_RANKS, which	25	
returns M	PI_PROC_NULL as the translat	jed rank.	26	
			27 28	
	UP_COMPARE(group1, group)	,	29	
IN	group1	first group (handle)	30	
IN	group2	second group (handle)	31	
OUT	result	result (integer)	32 33	
· ·			34	
int MPI_	Group_compare(MP1_Group g	roup1,MPI_Group group2, int *result)	35	
	p_compare(group1, group2,		36	
	(MPI_Group), INTENT(IN) : GER, INTENT(OUT) :: resu		37 38	
	GER, OPTIONAL, INTENT(OUT		39	
MPT CROIT		RESULT TERROR)	40	
	MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR) INTEGER GROUP1, GROUP2, RESULT, IERROR			
MPI IDEN	T results if the group members	and group order is exactly the same in both groups	42 43	
MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.				
			47	

```
6.3.2 Group Constructors
```

1

 $\mathbf{2}$

11

12

13

14

15

16 17

18

19

20

21 22 23

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.)

```
<sup>24</sup> MPI_COMM_GROUP(comm, group)
```

```
25
       IN
                 comm
                                            communicator (handle)
26
       OUT
                 group
                                             group corresponding to comm (handle)
27
28
     int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
29
30
     MPI_Comm_group(comm, group, ierror)
^{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
40
     MPI_GROUP_UNION(group1, group2, newgroup)
41
       IN
                                             first group (handle)
                 group1
42
       IN
                 group2
                                            second group (handle)
43
44
       OUT
                 newgroup
                                            union group (handle)
45
46
     int MPI_Group_union(MPI_Group group1, MPI_Group group2,
47
                    MPI_Group *newgroup)
48
```

<pre>MPI_Group_union(group1, group2, newgroup, ierror) TYPE(MPI_Group), INTENT(IN) :: group1, group2 TYPE(MPI_Group), INTENT(OUT) :: newgroup</pre>				
	MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR			
			7 8	
			° 9	
MPI_GR	OUP_INTERSECTION	l(group1, group2, newgroup)	10	
IN	group1	first group (handle)	11	
IN	group2	second group (handle)	12	
OUT	newgroup	intersection group (handle)	13 14	
001		monocolon group (nanalo)	15	
int MPI	_Group_intersection	n(MPI_Group group1, MPI_Group group2,	16	
	MPI_Group *n		17	
MDT Cro	un intergoction (ar	oup1, group2, newgroup, ierror)	18	
		NT(IN) :: group1, group2	19	
	-	NT(OUT) :: newgroup	20	
	-	TENT(OUT) :: ierror	21	
MDT (DO			22 23	
		OUP1, GROUP2, NEWGROUP, IERROR) 2, NEWGROUP, IERROR	20	
1111	Edent Gitoor 1, Gitoor		25	
			26	
MPI GR		roup1, group2, newgroup)	27	
IN		,	28	
	group1	first group (handle)	29	
IN	group2	second group (handle)	30	
OUT	newgroup	difference group (handle)	31 32	
			33	
int MPI	-	MPI_Group group1, MPI_Group group2,	34	
	MPI_Group *n	ewgroup)	35	
		p1, group2, newgroup, ierror)	36	
	_	NT(IN) :: group1, group2	37	
	-	NT(OUT) :: newgroup	38	
1N1	EGER, OPTIONAL, IN	TENT(OUT) :: ierror	39 40	
		P1, GROUP2, NEWGROUP, IERROR)	40	
INT	EGER GROUP1, GROUP	2, NEWGROUP, IERROR	42	
The set-	like operations are def	ined as follows:	43	
		st group (group1), followed by all elements of second group	44 45	
(gr	oup2) not in the first	group.	46	
	intersect all elements of the first group that are also in the second group, ordered as in the 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) 23TYPE(MPI_Group), INTENT(IN) :: group 24 INTEGER, INTENT(IN) :: n, ranks(n) 25TYPE(MPI_Group), INTENT(OUT) :: newgroup 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 27MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) 28INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR 2930 The function MPI_GROUP_INCL creates a group newgroup that consists of the 31 n processes in group with ranks $ranks[0], \ldots, ranks[n-1]$; the process with rank i in newgroup 32 is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid 33 rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, 34 then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder 35 the elements of a group. See also MPI_GROUP_COMPARE. 36 37 38MPI_GROUP_EXCL(group, n, ranks, newgroup) 39 IN group (handle) group 40IN number of elements in array ranks (integer) 41 n 42IN ranks array of integer ranks in group not to appear in 43 newgroup 44OUT newgroup new group derived from above, preserving the order 45defined by group (handle) 46 47 48

<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>			
<pre>MPI_Group_excl(group, n, ranks, newgroup, ierror) TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: n, ranks(n)</pre>			3 4 5 6
TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror			7 8
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR			9 10
		eates a group of processes newgroup that is obtained	11
by deleting	g from group those processes v	with ranks ranks[0] , ranks[n-1]. The ordering of	12 13
must be a	valid rank in group and all ele	ordering in group. Each of the n elements of ranks ements must be distinct; otherwise, the program is	$14 \\ 15$
erroneous.	If $n = 0$, then newgroup is identified to be a set of the set of	dentical to group.	16 17
MPL GRO	JP_RANGE_INCL(group, n, ra	nges newgroun)	18
IN	group	group (handle)	19
			20 21
IN	n	number of triplets in array ranges (integer)	21
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group	23
		of processes to be included in newgroup	24 25
OUT	newgroup	new group derived from above, in the order defined by	26
		ranges (handle)	27
int MDT (roup range incl(MPI Grouv	p group, int n, int ranges[][3],	28 29
IIIC MIL_(MPI_Group *newgroup)	group, int n, int langes[][3],	30
MPI Groun	_range_incl(group, n, ran	nges, newgroup, ierror)	31
	(MPI_Group), INTENT(IN) :		32
	ER, INTENT(IN) :: n, rai		$33 \\ 34$
	MPI_Group), INTENT(OUT)		35
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	36
MPI_GROUP	P_RANGE_INCL(GROUP, N, RAN	NGES, NEWGROUP, IERROR)	37
INTEC	ER GROUP, N, RANGES(3,*)	, NEWGROUP, IERROR	38
If ranges c	onsists of the triplets		39 40
(first)	$st_1, last_1, stride_1), \ldots, (first_n)$	$last_n, stride_n)$	41
then new a	roup consists of the sequence of	of processes in group with ranks	42
U			43 44
$first_1, first_1 + stride_1, \dots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \dots,$			45
1 act finat			46 47
$first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$			48

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

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

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

9 10

11

28

29 30

 31

32

40

45

12IN group (handle) group 13 IN n number of elements in array ranges (integer) 14a one-dimensional array of integer triplets of the form IN ranges 15(first rank, last rank, stride), indicating the ranks in 16group of processes to be excluded from the output 17 group newgroup. 18 19 OUT newgroup new group derived from above, preserving the order 20in group (handle) 2122int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], 23MPI_Group *newgroup) 24 MPI_Group_range_excl(group, n, ranges, newgroup, ierror) 25TYPE(MPI_Group), INTENT(IN) :: group 26INTEGER, INTENT(IN) :: n, ranges(3,n) 27

TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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.)

```
MPI_GROUP_FREE(group)
INOUT group group (handle)
int MPI_Group_free(MPI_Group *group)
MPI_Group_free(group, ierror)
    TYPE(MPI_Group), INTENT(INOUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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.)

```
36
                                                                                              37
6.4.1 Communicator Accessors
                                                                                              38
The following are all local operations.
                                                                                              39
                                                                                              40
                                                                                              41
MPI_COMM_SIZE(comm, size)
                                                                                              42
  IN
            comm
                                         communicator (handle)
                                                                                              43
                                                                                              44
  OUT
                                         number of processes in the group of comm (integer)
            size
                                                                                              45
                                                                                              46
int MPI_Comm_size(MPI_Comm comm, int *size)
                                                                                              47
                                                                                              48
MPI_Comm_size(comm, size, ierror)
```

```
Unofficial Draft for Comment Only
```

1	TYPE(MPI_Comm), INTENT(IN) ::	comm	
2	INTEGER, INTENT(OUT) :: size		
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
4	MDT COMM CITE (COMM CITE TEDDOD)		
5	MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR		
6	INTEGER COMM, SIZE, IERROR		
7			
8	-	valent to accessing the communicator's group with	
9	· · · · · · · · · · · · · · · · · · ·	, computing the size using MPI_GROUP_SIZE, and	
10		via MPI_GROUP_FREE. However, this function is	
11 12	so commonly used that this short	cut was introduced. (End of rationale.)	
12	Advice to users. This function	n indicates the number of processes involved in a	
14	communicator. For MPI_COMM_WORLD, it indicates the total number of processes		
15		rocesses has been changed by using the functions	
16	described in Chapter 10; note th	at the number of processes in MPI_COMM_WORLD	
17	does not change during the life of an MPI program.		
18	This call is often used with the	next call to determine the amount of concurrency	
19		program. The following call, MPI_COMM_RANK	
20	÷ 0	hat calls it in the range from $0 \dots \text{size} - 1$, where size	
21	is the return value of MPI_COMM	1_SIZE .(<i>End of advice to users.</i>)	
22			
23			
24 25	MPI_COMM_RANK(comm, rank)		
26	IN comm	communicator (handle)	
27	OUT rank	rank of the calling process in group of comm (integer)	
28			
29 30	<pre>int MPI_Comm_rank(MPI_Comm comm,</pre>	int *rank)	
31	<pre>MPI_Comm_rank(comm, rank, ierror)</pre>		
32	TYPE(MPI_Comm), INTENT(IN) ::		
33	INTEGER, INTENT(OUT) :: rank		
34	INTEGER, OPTIONAL, INTENT(OUT	C) :: ierror	
35	MPI_COMM_RANK(COMM, RANK, IERROR)		
36	INTEGER COMM, RANK, IERROR		
37	· · ·		
38 39	Rationale. This function is equi	valent to accessing the communicator's group with	
40	-), computing the rank using MPI_GROUP_RANK,	
41		oup via MPI_GROUP_FREE. However, this function	
42		ortcut was introduced. (End of rationale.)	
43		······································	
44	Advice to users. This function gi	ves the rank of the process in the particular commu-	
45	nicator's group. It is useful, as no	ted above, in conjunction with MPI_COMM_SIZE.	
46	Many programs will be written wi	th the master-slave model, where one process (such	
47	as the rank-zero process) will pla	ay a supervisory role, and the other processes will	
48	serve as compute nodes. In this	framework, the two preceding calls are useful for	

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

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

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

6.4.2 Communicator Constructors

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

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

This chapter presents the following communicator construction routines: MPI_COMM_CREATE, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO and MPI_COMM_SPLIT can be used to create both intracommunicators and intercommunicators; MPI_COMM_CREATE_GROUP and MPI_INTERCOMM_MERGE (see Section 6.6.2) can be used to create intracommunicators; and MPI_INTERCOMM_CREATE (see Section 6.6.2) can be used to create intercommunicators.

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

Unofficial Draft for Comment Only

 24

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

```
6
     MPI_COMM_DUP(comm, newcomm)
7
8
       IN
                                          communicator (handle)
                comm
9
       OUT
                                          copy of comm (handle)
                newcomm
10
11
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
12
13
    MPI_Comm_dup(comm, newcomm, ierror)
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
16
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                              ierror
17
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
18
         INTEGER COMM, NEWCOMM, IERROR
19
```

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

Advice to users. This operation is used to provide a parallel library with a duplicate
 communication space that has the same properties as the original communicator. This
 includes any attributes (see below) and topologies (see Chapter 7). This call is valid
 even if there are pending point-to-point communications involving the communicator
 comm. A typical call might involve a MPI_COMM_DUP at the beginning of the
 parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end
 of the call. Other models of communicator management are also possible.

This call applies to both intra- and inter-communicators. (End of advice to users.)

Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information. (End of advice to implementors.)

40 41 42

43

48

35

36 37

38

39

27

5

MPI_COMM_DUP_WITH_INFO(comm, info, newcomm)

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

6.4. COMMUNICATOR MANAGEMENT

1 int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm) 2 MPI_Comm_dup_with_info(comm, info, newcomm, ierror) 3 TYPE(MPI_Comm), INTENT(IN) :: comm 4 TYPE(MPI_Info), INTENT(IN) :: info 5 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR) 9 INTEGER COMM, INFO, NEWCOMM, IERROR 10 MPI_COMM_DUP_WITH_INFO behaves exactly as MPI_COMM_DUP except that the 11 hints provided by the argument info are associated with the output communicator newcomm. 1213 *Rationale.* It is expected that some hints will only be valid at communicator creation 14time. However, for legacy reasons, most communicator creation calls do not provide 15an info argument. One may associate info hints with a duplicate of any communicator 16 at creation time through a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) 17 18 19 MPI_COMM_IDUP(comm, newcomm, request) 2021IN comm communicator (handle) 22 OUT copy of comm (handle) newcomm 2324OUT communication request (handle) request 2526int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request) 27MPI_Comm_idup(comm, newcomm, request, ierror) 28 TYPE(MPI_Comm), INTENT(IN) :: comm 29 TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm 30 TYPE(MPI_Request), INTENT(OUT) :: request 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR) 34 INTEGER COMM, NEWCOMM, REQUEST, IERROR 35 MPI_COMM_IDUP is a nonblocking variant of MPI_COMM_DUP. With the exception 36 of its nonblocking behavior, the semantics of MPI_COMM_IDUP are as if MPI_COMM_DUP 37 was executed at the time that MPI_COMM_IDUP is called. For example, attributes changed 38 after MPI_COMM_IDUP will not be copied to the new communicator. All restrictions and 39 assumptions for nonblocking collective operations (see Section 5.12) apply to 40 MPI_COMM_IDUP and the returned request. 41 It is erroneous to use the communicator newcomm as an input argument to other MPI 42functions before the MPI_COMM_IDUP operation completes. 43 44 45464748 260 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
     MPI_COMM_IDUP_WITH_INFO(comm, info, newcomm, request)
2
       IN
                                           communicator (handle)
                comm
3
       IN
                info
                                           info object (handle)
4
5
       OUT
                newcomm
                                           copy of comm (handle)
6
       OUT
                                           communication request (handle)
                request
7
8
     int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,
9
                   MPI_Comm *newcomm, MPI_Request *request)
10
11
     MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Info), INTENT(IN) ::
                                           info
14
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
15
         TYPE(MPI_Request), INTENT(OUT) ::
                                               request
16
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                               ierror
17
     MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
18
         INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
19
20
         MPI_COMM_IDUP_WITH_INFO is a nonblocking variant of
21
     MPI_COMM_DUP_WITH_INFO. With the exception of its nonblocking behavior, the se-
22
     mantics of MPI_COMM_IDUP_WITH_INFO are as if MPI_COMM_DUP_WITH_INFO was
23
     executed at the time that MPI_COMM_IDUP_WITH_INFO is called. For example, attributes
24
     or info hints changed after MPI_COMM_IDUP_WITH_INFO will not be copied to the new
25
     communicator. All restrictions and assumptions for nonblocking collective operations (see
26
     Section 5.12) apply to MPI_COMM_IDUP_WITH_INFO and the returned request.
27
         It is erroneous to use the communicator newcomm as an input argument to other MPI
28
     functions before the MPI_COMM_IDUP_WITH_INFO operation completes.
29
30
                      The MPI_COMM_IDUP and MPI_COMM_IDUP_WITH_INFO functions
          Rationale.
^{31}
          are crucial for the development of purely nonblocking libraries (see [36]). (End of
32
          rationale.)
33
34
35
     MPI_COMM_CREATE(comm, group, newcomm)
36
37
       IN
                comm
                                           communicator (handle)
38
       IN
                                           group, which is a subset of the group of comm (handle)
                group
39
       OUT
                newcomm
                                           new communicator (handle)
40
41
     int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
42
43
     MPI_Comm_create(comm, group, newcomm, ierror)
44
         TYPE(MPI_Comm), INTENT(IN) ::
                                           comm
45
         TYPE(MPI_Group), INTENT(IN) ::
                                            group
46
         TYPE(MPI_Comm), INTENT(OUT) ::
                                            newcomm
47
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                               ierror
48
```

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR) INTEGER COMM, GROUP, NEWCOMM, IERROR

If comm is an intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm. Each process must call MPI_COMM_CREATE with a group argument that is a subgroup 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 then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

Rationale. The interface supports the original mechanism from MPI-1.1, which required the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI_COMM_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
- It permits implementations to sometimes avoid communication related to context creation.

(End of rationale.)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE, can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) to further subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

41 Advice to implementors. When calling MPI_COMM_DUP, all processes call with the 42same group (the group associated with the communicator). When calling MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups. 43 44For 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 4546of a larger context name space and reduces error checking. Implementations may 47strike various compromises between these conflicting goals, such as bulk allocation of 48 multiple contexts in one collective operation.

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

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

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Fig- $\overline{7}$ ure 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.*)

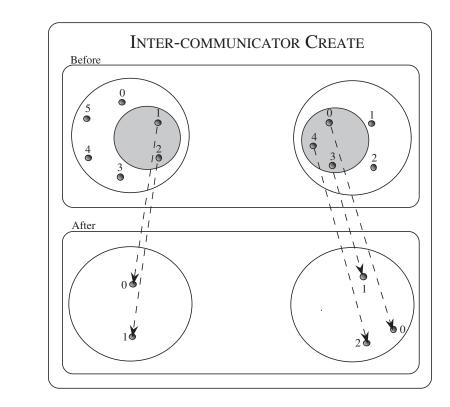


Figure 6.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

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.

```
1
        MPI_Comm inter_comm, new_inter_comm;
                                                                                     2
        MPI_Group local_group, group;
                                                                                     3
                   rank = 0; /* rank on left side to include in
        int
                                 new inter-comm */
                                                                                     4
                                                                                     5
                                                                                     6
        /* Construct the original intercommunicator: "inter_comm" */
                                                                                     7
                                                                                     8
                                                                                     9
        /* Construct the group of processes to be in new
                                                                                    10
            intercommunicator */
                                                                                    11
        if (/* I'm on the left side of the intercommunicator */) {
          MPI_Comm_group ( inter_comm, &local_group );
                                                                                    12
          MPI_Group_incl ( local_group, 1, &rank, &group );
                                                                                    13
          MPI_Group_free ( &local_group );
                                                                                    14
                                                                                    15
        }
                                                                                    16
        else
                                                                                    17
          MPI_Comm_group ( inter_comm, &group );
                                                                                    18
                                                                                    19
        MPI_Comm_create ( inter_comm, group, &new_inter_comm );
        MPI_Group_free( &group );
                                                                                    20
                                                                                    21
                                                                                    22
                                                                                    23
MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)
                                                                                    24
 IN
           comm
                                     intracommunicator (handle)
                                                                                    25
                                                                                    26
 IN
                                     group, which is a subset of the group of comm (handle)
           group
                                                                                    27
                                     tag (integer)
 IN
          tag
                                                                                    28
 OUT
           newcomm
                                     new communicator (handle)
                                                                                    29
                                                                                    30
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
                                                                                    31
              MPI_Comm *newcomm)
                                                                                    32
                                                                                    33
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
                                                                                    34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    35
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                    36
    INTEGER, INTENT(IN) :: tag
                                                                                    37
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                    38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    39
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
                                                                                    40
                                                                                    41
    INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
                                                                                    42
    MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however,
                                                                                    43
MPI_COMM_CREATE must be called by all processes in the group of
                                                                                    44
comm, whereas MPI_COMM_CREATE_GROUP must be called by all processes in group,
                                                                                    45
which is a subgroup of the group of comm. In addition, MPI_COMM_CREATE_GROUP
                                                                                    46
requires that comm is an intracommunicator. MPI_COMM_CREATE_GROUP returns a new
                                                                                    47
intracommunicator, newcomm, for which the group argument defines the communication
```

1 group. No cached information propagates from comm to newcomm. Each process must $\mathbf{2}$ provide a group argument that is a subgroup of the group associated with comm; this 3 could be MPI_GROUP_EMPTY. If a non-empty group is specified, then all processes in that 4 group must call the function, and each of these processes must provide the same arguments, 5including a group that contains the same members with the same ordering. Otherwise 6 the call is erroneous. If the calling process is a member of the group given as the group 7 argument, then newcomm is a communicator with group as its associated group. If the 8 calling process is not a member of group, e.g., group is MPI_GROUP_EMPTY, then the call 9 is a local operation and MPI_COMM_NULL is returned as newcomm. 10 Rationale. Functionality similar to MPI_COMM_CREATE_GROUP can be imple-11 mented through repeated MPI_INTERCOMM_CREATE and 12MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communica-13 tors at each process in group and build up an intracommunicator with group 14 group [16]. Such an algorithm requires the creation of many intermediate communi-15cators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation 16that avoids this overhead. (End of rationale.) 17 18 Advice to users. An intercommunicator can be created collectively over processes in 19 the union of the local and remote groups by creating the local communicator using 20MPI_COMM_CREATE_GROUP and using that communicator as the local communi-21cator argument to MPI_INTERCOMM_CREATE. (End of advice to users.) 22 23The tag argument does not conflict with tags used in point-to-point communication and 24 is not permitted to be a wildcard. If multiple threads at a given process perform concurrent 25MPI_COMM_CREATE_GROUP operations, the user must distinguish these operations by 26providing different tag or comm arguments. 2728MPI_COMM_CREATE may provide lower overhead than Advice to users. 29 MPI_COMM_CREATE_GROUP because it can take advantage of collective communi-30 cation on comm when constructing newcomm. (End of advice to users.) 31 32 33 MPI_COMM_SPLIT(comm, color, key, newcomm) 34 IN comm communicator (handle) 35 36 IN color control of subset assignment (integer) 37 IN control of rank assignment (integer) key 38 OUT newcomm new communicator (handle) 39 40 41 int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm) 42MPI_Comm_split(comm, color, key, newcomm, ierror) 43 TYPE(MPI_Comm), INTENT(IN) :: comm 44 INTEGER, INTENT(IN) :: color, key 45TYPE(MPI_Comm), INTENT(OUT) :: newcomm 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47

⁴⁸ MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)

INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in **newcomm**. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all processes that are not members of their group argument provide $color = MPI_UNDEFINED$.

The value of color must be non-negative or MPI_UNDEFINED.

This is an extremely powerful mechanism for dividing a single Advice to users. communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a 20hierarchy of computations, such as for multigrid, or linear algebra. For intracommu-21nicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to 22split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful 23when some processes do not have complete information of the other members in their group, but all processes know (the color of) the group to which they belong. In this case, the MPI implementation discovers the other group members via communication. MPI_COMM_CREATE is useful when all processes have complete information of the 27members of their group. In this case, MPI can avoid the extra communication required to discover group membership. MPI_COMM_CREATE_GROUP is useful when all pro-29cesses in a given group have complete information of the members of their group and 30 synchronization with processes outside the group can be avoided.

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

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

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

Rationale. color is restricted to be non-negative, so as not to confict with the value assigned to MPI_UNDEFINED. (End of rationale.)

The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercom-

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

 24

25

26

28

 31

32

33

34

35

36

37

38

39

40

41

42

43 44

45

46 47

¹ municator. The key argument describes the relative rank of processes on each side of the
 ² intercommunicator (see Figure 6.2). For those colors that are specified only on one side of
 ³ the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned
 ⁴ to those processes that specify MPI_UNDEFINED as the color.

5

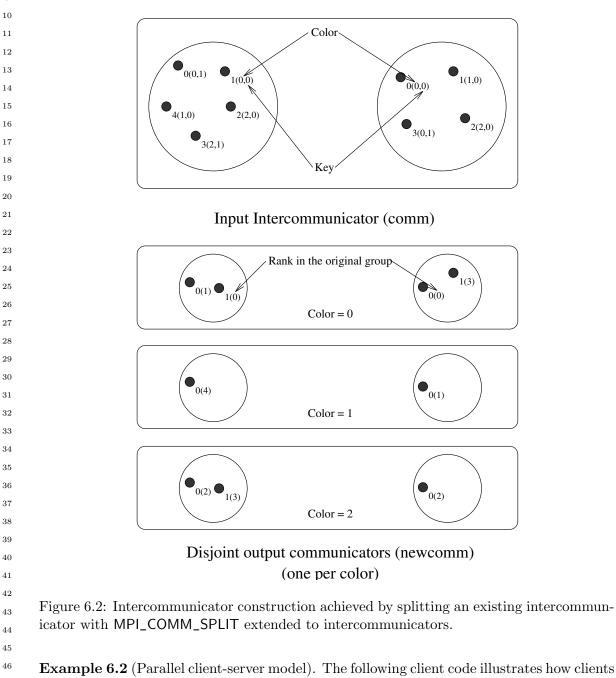
6

7

8

9

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 inustrates now 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.

```
1
        /* Client code */
                                                                                        \mathbf{2}
        MPI_Comm multiple_server_comm;
                                                                                        3
        MPI_Comm single_server_comm;
                    color, rank, num_servers;
                                                                                        4
        int
                                                                                        5
                                                                                        6
         /* Create intercommunicator with clients and servers:
            multiple_server_comm */
                                                                                        7
         . . .
                                                                                        9
                                                                                        10
        /* Find out the number of servers available */
                                                                                        11
        MPI_Comm_remote_size ( multiple_server_comm, &num_servers );
                                                                                        12
        /* Determine my color */
                                                                                        13
        MPI_Comm_rank ( multiple_server_comm, &rank );
                                                                                        14
                                                                                        15
        color = rank % num_servers;
                                                                                        16
                                                                                        17
        /* Split the intercommunicator */
                                                                                        18
        MPI_Comm_split ( multiple_server_comm, color, rank,
                                                                                        19
                           &single_server_comm );
                                                                                        20
The following is the corresponding server code:
                                                                                        21
        /* Server code */
                                                                                        22
                                                                                        23
        MPI_Comm multiple_client_comm;
                                                                                        24
        MPI_Comm single_server_comm;
                                                                                        25
                   rank;
        int
                                                                                        26
                                                                                        27
         /* Create intercommunicator with clients and servers:
                                                                                        28
            multiple_client_comm */
                                                                                        29
         . . .
                                                                                        30
                                                                                        31
        /* Split the intercommunicator for a single server per group
                                                                                        32
            of clients */
                                                                                        33
        MPI_Comm_rank ( multiple_client_comm, &rank );
                                                                                        34
        MPI_Comm_split ( multiple_client_comm, rank, 0,
                                                                                        35
                           &single_server_comm );
                                                                                        36
                                                                                        37
                                                                                        38
MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)
                                                                                        39
 IN
                                      communicator (handle)
           comm
                                                                                        40
 IN
                                      type of processes to be grouped together (integer)
                                                                                        41
           split_type
                                                                                        42
 IN
           key
                                      control of rank assignment (integer)
                                                                                        43
 IN
           info
                                      info argument (handle)
                                                                                        44
 OUT
           newcomm
                                      new communicator (handle)
                                                                                        45
                                                                                        46
                                                                                        47
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
                                                                                        48
              MPI_Info info, MPI_Comm *newcomm)
```

```
1
     MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
\mathbf{2}
          TYPE(MPI_Comm), INTENT(IN) :: comm
3
          INTEGER, INTENT(IN) :: split_type, key
4
          TYPE(MPI_Info), INTENT(IN) :: info
5
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
8
          INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
9
10
     This function partitions the group associated with comm into disjoint subgroups, based on
11
     the type specified by split_type. Each subgroup contains all processes of the same type.
12
     Within each subgroup, the processes are ranked in the order defined by the value of the
13
     argument key, with ties broken according to their rank in the old group. A new commu-
14
     nicator is created for each subgroup and returned in newcomm. This is a collective call;
15
     all processes must provide the same split_type, but each process is permitted to provide
16
     different values for key. An exception to this rule is that a process may supply the type
17
     value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL.
18
          The following type is predefined by MPI:
19
      MPI_COMM_TYPE_SHARED — this type splits the communicator into subcommunicators,
20
           each of which can create a shared memory region.
21
22
                                      Implementations can define their own types, or use the
           Advice to implementors.
23
           info argument, to assist in creating communicators that help expose platform-specific
^{24}
           information to the application. (End of advice to implementors.)
25
26
     6.4.3 Communicator Destructors
27
28
29
     MPI_COMM_FREE(comm)
30
       INOUT
                 comm
                                             communicator to be destroyed (handle)
^{31}
32
33
     int MPI_Comm_free(MPI_Comm *comm)
34
     MPI_Comm_free(comm, ierror)
35
          TYPE(MPI_Comm), INTENT(INOUT) ::
                                                 comm
36
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
37
38
     MPI_COMM_FREE(COMM, IERROR)
39
          INTEGER COMM, IERROR
40
          This collective operation marks the communication object for deallocation. The handle
41
     is set to MPI_COMM_NULL. Any pending operations that use this communicator will com-
42
     plete normally; the object is actually deallocated only if there are no other active references
43
     to it. This call applies to intra- and inter-communicators. The delete callback functions for
44
     all cached attributes (see Section 6.7) are called in arbitrary order.
45
46
           Advice to implementors. Though collective, it is anticipated that this operation will
47
           normally be implemented to be local, though a debugging version of an MPI library
```

Unofficial Draft for Comment Only

might choose to synchronize. (End of advice to implementors.)

6.4.4 Communicator Info

Hints specified via info (see Chapter 9) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or minimize use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_SPLIT_TYPE, MPI_DIST_GRAPH_CREATE, and MPI_DIST_GRAPH_CREATE_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

Info hints are not propagated by MPI from one communicator to another. The following info keys are valid for all communicators.

- mpi_assert_no_any_tag (boolean, default: false): If set to true, then the implementation
 may assume that the process will not use the MPI_ANY_TAG wildcard on the given
 communicator.
- mpi_assert_no_any_source (boolean, default: false): If set to true, then the implementation
 may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given
 communicator.
- mpi_assert_exact_length (boolean, default: false): If set to true, then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.
- mpi_assert_allow_overtaking (boolean, default: false): If set to true, then the implementation may assume that point-to-point communications on the given communicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

Advice to users. Use of the mpi_assert_allow_overtaking info key can result in nondeterminism in the message matching order. (*End of advice to users.*)

Advice to users. Some optimizations may only be possible when all processes in the 46 group of the communicator provide a given info key with the same value. (End of 47 advice to users.) 48

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

1 MPI_COMM_SET_INFO(comm, info) 2 INOUT comm communicator (handle) 3 IN info info object (handle) 4 56 int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info) 7 MPI_Comm_set_info(comm, info, ierror) 8 TYPE(MPI_Comm), INTENT(IN) :: comm 9 TYPE(MPI_Info), INTENT(IN) :: info 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 12MPI_COMM_SET_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR 13 14MPI_COMM_SET_INFO updates the hints of the communicator associated with comm 15using the hints provided in info. This operation has no effect on previously set or defaulted 16hints that are not specified by info. It also has no effect on previously set or defaulted 17hints that are specified by info, but are ignored by the MPI implementation in this call to 18 MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object 19 may be different on each process, but any info entries that an implementation requires to 20be the same on all processes must appear with the same value in each process's info object. 2122Some info items that an implementation can use when it creates Advice to users. 23a communicator cannot easily be changed once the communicator has been created. 24Thus, an implementation may ignore hints issued in this call that it would have 25accepted in a creation call. An implementation may also be unable to update certain 26info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to 27determine whether updates to existing info hints were ignored by the implementation. 28(End of advice to users.) 29 30 Advice to users. Setting info hints on the predefined communicators 31MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to 32 these global objects may affect all components of the application, including libraries 33 and tools. Users must ensure that all components of the application that use a given 34 communicator, including libraries and tools, can comply with any info hints associated 35with that communicator. (End of advice to users.) 36 37 38 MPI_COMM_GET_INFO(comm, info_used) 39 40IN communicator object (handle) comm 41 OUT info_used new info object (handle) 4243 int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used) 4445MPI_Comm_get_info(comm, info_used, ierror) 46TYPE(MPI_Comm), INTENT(IN) :: comm 47 TYPE(MPI_Info), INTENT(OUT) :: info_used 48 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_GET_INFO(COMM, INFO_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 related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. 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.

```
6.5 Motivating Examples
```

```
6.5.1 Current Practice #1
```

Example #1a:

```
int main(int argc, char *argv[])
{
    int me, size;
    ...
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank (MPI_COMM_WORLD, &me);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    (void)printf ("Process %d size %d\n", me, size);
    ...
    MPI_Finalize();
    return 0;
}
```

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):

```
int main(int argc, char *argv[])
{
    int me, size;
    int SOME_TAG = 0;
    ...
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
    if((me % 2) == 0)
    {
        /* send unless highest-numbered process */
    }
}
```

Unofficial Draft for Comment Only

 $\mathbf{2}$

```
1
                 if((me + 1) < size)
\mathbf{2}
                    MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
3
             }
4
             else
5
                 MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
6
7
              . . .
8
             MPI_Finalize();
9
             return 0;
10
          }
11
     Example #1b schematically illustrates message exchanges between "even" and "odd" pro-
12
     cesses in the "all" communicator.
13
14
     6.5.2 Current Practice #2
15
16
         int main(int argc, char *argv[])
17
         {
18
           int me, count;
19
           void *data;
20
           . . .
21
22
           MPI_Init(&argc, &argv);
23
           MPI_Comm_rank(MPI_COMM_WORLD, &me);
^{24}
25
           if(me == 0)
26
           {
27
                /* get input, create buffer ''data'' */
28
                . . .
29
           }
30
31
           MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
32
33
           . . .
34
           MPI_Finalize();
35
           return 0;
36
         }
37
38
     This example illustrates the use of a collective communication.
39
40
            (Approximate) Current Practice #3
     6.5.3
41
42
        int main(int argc, char *argv[])
        {
43
          int me, count, count2;
44
          void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
45
46
          MPI_Group group_world, grprem;
47
          MPI_Comm commslave;
          static int ranks[] = {0};
48
```

```
. . .
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
                                                                                  5
                                                                                  6
  MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);
                                                                                  9
  if(me != 0)
                                                                                  10
  {
                                                                                  11
    /* compute on slave */
                                                                                  12
    MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commslave);
                                                                                  13
                                                                                  14
                                                                                  15
    MPI_Comm_free(&commslave);
                                                                                  16
  }
                                                                                  17
  /* zero falls through immediately to this reduce, others do later... */
                                                                                  18
  MPI_Reduce(send_buf2, recv_buf2, count2,
                                                                                  19
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
                                                                                  20
                                                                                  21
  MPI_Group_free(&group_world);
  MPI_Group_free(&grprem);
                                                                                  22
                                                                                  23
  MPI_Finalize();
                                                                                  24
  return 0;
                                                                                  25
}
                                                                                  26
                                                                                  27
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commslave) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

Example #46.5.4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
#define TAG ARBITRARY 12345
#define SOME_COUNT
                          50
int main(int argc, char *argv[])
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
```

1

 $\mathbf{2}$

3

4

7 8

28

29

30

31

32

33

34 35

36 37

38

39

40 41

42

43 44

45

46

47

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

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

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

```
}
     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
     MPI_Group_free(&group_world);
     MPI_Finalize();
     return 0;
   }
The library:
   void lib_call(MPI_Comm comm)
   Ł
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
        }
     else
     {
       /* work */
       MPI_Send(..., 0, ARBITRARY_TAG, comm);
     }
#ifdef EXAMPLE_2C
     /* include (resp, exclude) for safety (resp, no safety): */
     MPI_Barrier(comm);
#endif
   }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [57]). Here we rely on two guarantees of MPI: pairwise ordering of 48

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 41

42

43

1 messages between processes in the same context, and source selectivity — deleting either $\mathbf{2}$ feature removes the guarantee that back-masking cannot be required.

3 Algorithms that try to do non-deterministic broadcasts or other calls that include wild-4 card operations will not generally have the good properties of the deterministic implemen- $\mathbf{5}$ tations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize 6 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.

1112

1314

15

16

41

 $\overline{7}$

8

9

10

Inter-Communication 6.6

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.

17All communication described thus far has involved communication between processes 18 that are members of the same group. This type of communication is called "intra-com-19munication" and the communicator used is called an "intra-communicator," as we have 20noted earlier in the chapter. 21

In modular and multi-disciplinary applications, different process groups execute distinct 22modules and processes within different modules communicate with one another in a pipeline 23or a more general module graph. In these applications, the most natural way for a process 24 to specify a target process is by the rank of the target process within the target group. In 25applications that contain internal user-level servers, each server may be a process group that 26provides services to one or more clients, and each client may be a process group that uses the 27services of one or more servers. It is again most natural to specify the target process by rank 28within the target group in these applications. This type of communication is called "int-29er-communication" and the communicator used is called an "inter-communicator," as 30 introduced earlier. 31

An inter-communication is a point-to-point communication between processes in 32 different groups. The group containing a process that initiates an inter-communication 33 operation is called the "local group," that is, the sender in a send and the receiver in a 34receive. The group containing the target process is called the "remote group," that is, 35 the receiver in a send and the sender in a receive. As in intra-communication, the target 36 process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank 37 is relative to a second, remote group.

38 All inter-communicator constructors are blocking except for MPI_COMM_IDUP and 39 require that the local and remote groups be disjoint. 40

Advice to users. The groups must be disjoint for several reasons. Primarily, this 42is the intent of the intercommunicators — to provide a communicator for commu-43 nication between disjoint groups. This is reflected in the definition of

44MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the pro-45cesses in the created intracommunicator; this ranking makes little sense if the groups 46are not disjoint. In addition, the natural extension of collective operations to inter-47 communicators makes the most sense when the groups are disjoint. (End of advice to 48 users.)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both interand intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group send_context receive_context source

For inter-communicators, group describes the remote group, and source is the rank of the process in the local group. For intra-communicators, group is the communicator group (remote=local), source is the rank of the process in this group, and send context and receive context are identical. A group can be represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process \mathbf{P} in group \mathcal{P} , which has an intercommunicator $\mathbf{C}_{\mathcal{P}}$, and a process \mathbf{Q} in group \mathcal{Q} , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $C_{\mathcal{P}}$.group describes the group \mathcal{Q} and $C_{\mathcal{Q}}$.group describes the group \mathcal{P} .
- $C_{\mathcal{P}}$.send_context = $C_{\mathcal{Q}}$.receive_context and the context is unique in \mathcal{Q} ; $C_{\mathcal{P}}$.receive_context = $C_{\mathcal{Q}}$.send_context and this context is unique in \mathcal{P} .
- $C_{\mathcal{P}}$.source is rank of **P** in \mathcal{P} and $C_{\mathcal{Q}}$.source is rank of **Q** in \mathcal{Q} .

Assume that \mathbf{P} sends a message to \mathbf{Q} using the inter-communicator. Then \mathbf{P} uses the **group** table to find the absolute address of \mathbf{Q} ; **source** and **send_context** are appended to the message.

Assume that \mathbf{Q} posts a receive with an explicit source argument using the intercommunicator. Then \mathbf{Q} matches **receive_context** to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

Unofficial Draft for Comment Only

 24

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) OUT flag (logical) int MPI_Comm_test_inter(MPI_Comm comm, int *flag) MPI_Comm_test_inter(comm, flag, ierror) 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.

Μ	PI_COMM_SIZE	returns the size of the local group.
M	PI_COMM_GROUP	returns the local group.
M	PI_COMM_RANK	returns the rank in the local group

33 34 35

1

 $\mathbf{2}$

3

4

5 6

7 8 9

10

11

19 20

21

22

23

27

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both
 communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results.
 Both corresponding local and remote groups must compare correctly to get the results
 MPI_CONGRUENT or MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result
 because either the local or remote groups were similar but not identical.

⁴¹ The following accessors provide consistent access to the remote group of an inter-⁴² communicator. The following are all local operations.

 $44 \\ 45$

- 40
- 46
- 47 48

MPI_COMM_REMOTE_SIZE(comm, size) 1 $\mathbf{2}$ IN comm inter-communicator (handle) 3 OUT number of processes in the remote group of comm size 4 (integer) 56 int MPI_Comm_remote_size(MPI_Comm comm, int *size) 7 8 MPI_Comm_remote_size(comm, size, ierror) 9 TYPE(MPI_Comm), INTENT(IN) :: comm 10 INTEGER, INTENT(OUT) :: size 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) 13 INTEGER COMM, SIZE, IERROR 14151617 MPI_COMM_REMOTE_GROUP(comm, group) 18 IN inter-communicator (handle) comm 19 OUT remote group corresponding to comm (handle) group 202122 int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group) 23MPI_Comm_remote_group(comm, group, ierror) 24 TYPE(MPI_Comm), INTENT(IN) :: comm 25TYPE(MPI_Group), INTENT(OUT) :: group 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728 MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR) 29 INTEGER COMM, GROUP, IERROR 30 31 Rationale. Symmetric access to both the local and remote groups of an inter-32 communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE 33 have been provided. (End of rationale.) 34 356.6.2 Inter-communicator Operations 36 37 This section introduces four blocking inter-communicator operations. 38 MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-com-39 municator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merg-40 ing the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP 41 and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, 42respectively.

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.)

Unofficial Draft for Comment Only

43

44

45

46

47

1 The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator $\mathbf{2}$ from two existing intra-communicators, in the following situation: At least one selected 3 member from each group (the "group leader") has the ability to communicate with the 4 selected member from the other group; that is, a "peer" communicator exists to which both $\mathbf{5}$ leaders belong, and each leader knows the rank of the other leader in this peer communicator. 6 Furthermore, members of each group know the rank of their leader. 7Construction of an inter-communicator from two intra-communicators requires separate 8 collective operations in the local group and in the remote group, as well as a point-to-point 9 communication between a process in the local group and a process in the remote group. 10 In standard MPI implementations (with static process allocation at initialization), the 11MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this 12peer communicator. For applications that have used spawn or join, it may be necessary to 13 first create an intracommunicator to be used as peer. 14The application topology functions described in Chapter 7 do not apply to inter-15communicators. Users that require this capability should utilize 16MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or carte-17sian topology capabilities to that intra-communicator, creating an appropriate topology-18 oriented intra-communicator. Alternatively, it may be reasonable to devise one's own ap-19plication topology mechanisms for this case, without loss of generality. 2021MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, 22 newintercomm) 23 24 IN local_comm local intra-communicator (handle) 25IN local_leader rank of local group leader in local_comm (integer) 26IN "peer" communicator; significant only at the peer_comm 27local_leader (handle) 2829IN remote_leader rank of remote group leader in peer_comm; significant 30 only at the local_leader (integer) 31IN tag (integer) tag 32 OUT newintercomm new inter-communicator (handle) 33 34int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, 35 36 MPI_Comm peer_comm, int remote_leader, int tag, 37 MPI_Comm *newintercomm) 38 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, 39 tag, newintercomm, ierror) 40 TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm 41 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag 42TYPE(MPI_Comm), INTENT(OUT) :: newintercomm 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4445MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, 46TAG, NEWINTERCOMM, IERROR) 47 INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, 48 NEWINTERCOMM, IERROR

This call creates an inter-communicator. It is collective over the union of the local and remote groups. Processes should provide identical local_comm and local_leader arguments within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)		
IN	intercomm	Inter-Communicator (handle)
IN	high	(logical)
OUT	newintracomm	new intra-communicator (handle)
<pre>int MPI_Intercomm_merge(MPI_Comm intercomm, int high,</pre>		
<pre>MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: intercomm</pre>		
LOGICAL, INTENT(IN) :: high		
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH		

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group before the "high" group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

Advice to implementors. The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE, and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

6.6.3 Inter-Communication Examples

Example 1: Three-Group "Pipeline"

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

```
int main(int argc, char *argv[])
```

 $\mathbf{2}$

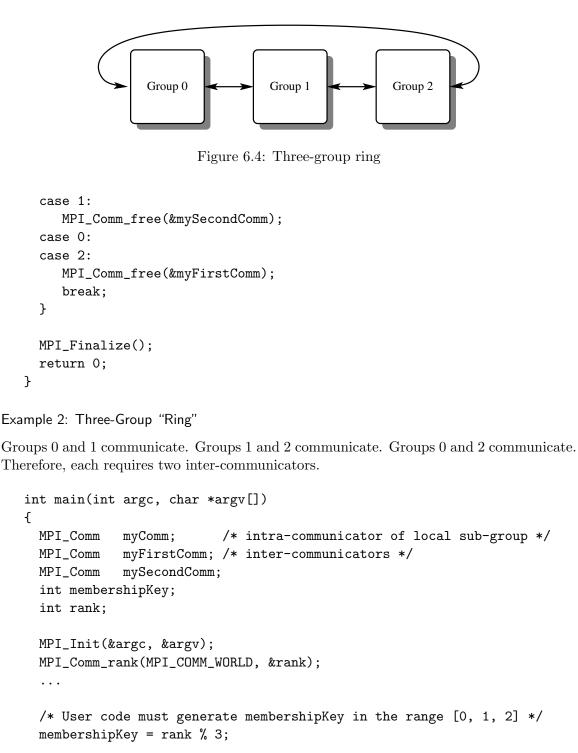
 24

```
1
2
3
                      Group 0
                                          Group 1
                                                               Group 2
4
5
6
7
                               Figure 6.3: Three-group pipeline
8
9
        {
10
          MPI_Comm
                      myComm;
                                      /* intra-communicator of local sub-group */
11
          MPI_Comm
                      myFirstComm; /* inter-communicator */
12
                      mySecondComm; /* second inter-communicator (group 1 only) */
          MPI_Comm
13
          int membershipKey;
14
          int rank;
15
16
          MPI_Init(&argc, &argv);
17
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18
19
          /* User code must generate membershipKey in the range [0, 1, 2] */
20
          membershipKey = rank % 3;
21
22
          /* Build intra-communicator for local sub-group */
23
          MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
24
25
          /* Build inter-communicators. Tags are hard-coded. */
26
          if (membershipKey == 0)
27
          ſ
                                  /* Group 0 communicates with group 1. */
28
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
29
                                   1, &myFirstComm);
30
          }
31
          else if (membershipKey == 1)
32
          ſ
                           /* Group 1 communicates with groups 0 and 2. */
33
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
34
                                    1, &myFirstComm);
35
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
36
                                   12, &mySecondComm);
37
          }
38
          else if (membershipKey == 2)
39
          {
                                  /* Group 2 communicates with group 1. */
40
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
41
                                   12, &myFirstComm);
42
          }
43
44
          /* Do work ... */
45
46
          switch(membershipKey) /* free communicators appropriately */
47
          ſ
48
```

}

}

{



```
/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
/* Build inter-communicators. Tags are hard-coded. */
if (membershipKey == 0)
{
              /* Group 0 communicates with groups 1 and 2. */
 MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
```

Unofficial Draft for Comment Only

 24

 31

1	1, &myFirstComm);
2	MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
3	2, &mySecondComm);
4	}
5	else if (membershipKey == 1)
6	<pre>{ /* Group 1 communicates with groups 0 and 2. */</pre>
7	MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
8	1, &myFirstComm);
9	MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
10	12, &mySecondComm);
11	}
12	else if (membershipKey == 2)
13	<pre>{ /* Group 2 communicates with groups 0 and 1. */</pre>
14	MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
15	2, &myFirstComm);
16	MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
17	12, &mySecondComm);
18	}
19	
20	/* Do some work */
21	
22	<pre>/* Then free communicators before terminating */</pre>
23	<pre>MPI_Comm_free(&myFirstComm);</pre>
24	<pre>MPI_Comm_free(&mySecondComm);</pre>
25	<pre>MPI_Comm_free(&myComm);</pre>
26	<pre>MPI_Finalize();</pre>
27	return 0;
28	}
29	
0.0	

6.7 Caching

30

31 32

33

34

35

36

37

38 39

40

 41

42

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

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

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attribute-caching mechanism. (*End of advice to* users.)

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

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI_ADDRESS_KIND.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

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,45in response to explicit application requests.This avoids problems that result from re-46peated crossings between user and system space.(This synchronous calling rule is a47general property of MPI.)48

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

1 The choice of key values is under control of MPI. This allows MPI to optimize its $\mathbf{2}$ implementation of attribute sets. It also avoids conflict between independent modules 3 caching information on the same communicators. 4 A much smaller interface, consisting of just a callback facility, would allow the entire 5caching facility to be implemented by portable code. However, with the minimal call-6 back interface, some form of table searching is implied by the need to handle arbitrary 7 communicators. In contrast, the more complete interface defined here permits rapid 8 access to attributes through the use of pointers in communicators (to find the attribute 9 table) and cleverly chosen key values (to retrieve individual attributes). In light of the 10 efficiency "hit" inherent in the minimal interface, the more complete interface defined 11 here is seen to be superior. (End of advice to implementors.) 1213MPI provides the following services related to caching. They are all process local. 14156.7.2 Communicators 1617Functions for caching on communicators are: 18 19MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, 20extra_state) 2122IN comm_copy_attr_fn copy callback function for comm_keyval (function) 23IN comm_delete_attr_fn delete callback function for comm_keyval (function) 24OUT comm_keyval key value for future access (integer) 2526IN extra state for callback functions extra_state 2728int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, 29MPI_Comm_delete_attr_function *comm_delete_attr_fn, 30 int *comm_keyval, void *extra_state) 31 MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, 32 extra_state, ierror) 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 39 MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL, 40EXTRA_STATE, IERROR) 41 EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN 42INTEGER COMM_KEYVAL, IERROR 43 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 44Generates a new attribute key. Keys are locally unique in a process, and opaque to 45user, though they are explicitly stored in integers. Once allocated, the key value can be 46used to associate attributes and access them on any locally defined communicator. 47

48 The C callback functions are:

```
1
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                                                                     \mathbf{2}
              void *extra_state, void *attribute_val_in,
                                                                                     3
              void *attribute_val_out, int *flag);
                                                                                     4
and
                                                                                     5
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                                                                     6
              void *attribute_val, void *extra_state);
                                                                                     7
                                                                                     8
which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
                                                                                     9
With the mpi_f08 module, the Fortran callback functions are:
                                                                                     10
ABSTRACT INTERFACE
                                                                                     11
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
                                                                                     12
                                                                                     13
      TYPE(MPI_Comm) :: oldcomm
                                                                                     14
      INTEGER :: comm_keyval, ierror
                                                                                     15
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     16
      attribute_val_out
                                                                                     17
      LOGICAL :: flag
                                                                                     18
and
                                                                                     19
ABSTRACT INTERFACE
                                                                                     20
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
                                                                                     21
  attribute_val, extra_state, ierror)
                                                                                     22
      TYPE(MPI_Comm) :: comm
                                                                                     23
      INTEGER :: comm_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 COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                                                                                     28
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     29
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
                                                                                     30
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     31
               ATTRIBUTE_VAL_OUT
                                                                                     32
    LOGICAL FLAG
                                                                                     33
and
                                                                                     34
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                                                                                     35
              EXTRA_STATE, IERROR)
                                                                                     36
    INTEGER COMM, COMM_KEYVAL, IERROR
                                                                                     37
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     38
                                                                                     39
    The comm_copy_attr_fn function is invoked when a communicator is duplicated by
                                                                                     40
MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type
                                                                                     41
MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
                                                                                     42
```

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).

47 48

43

44

45

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

1 The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN $\mathbf{2}$ or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a 3 function that does nothing other than returning flag = 0 or .FALSE. (depending on whether 4 the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and $\mathbf{5}$ MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or 6 .TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. 7These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, 8 whose use is deprecated. 9

- 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.
- ¹⁵ A valid copy function is one that completely duplicates the information by making ¹⁶ a full duplicate copy of the data structures implied by an attribute; another might ¹⁷ just make another reference to that data structure, while using a reference-count ¹⁸ mechanism. Other types of attributes might not copy at all (they might be specific ¹⁹ 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.
- ²⁸ 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).
- ³³ The argument comm_delete_attr_fn may be specified as

21

22

23

24

44

- ³⁴ MPI_COMM_NULL_DELETE_FN from either C or Fortran.
- ³⁵ MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning
- MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose
 use is deprecated.
- ³⁸ If an attribute copy function or attribute delete function returns other than
- MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE),
 is erroneous.
- ⁴¹ The special key value MPI_KEYVAL_INVALID is never returned by
- MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key
 values.
- ⁴⁵ Advice to implementors. The predefined Fortran functions
- ⁴⁶ MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
- 47 MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and
- the mpi_f08 module with the same name, but with different interfaces. Each function

can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other routine within mpi_f08 declared with CONTAINS. These routines have different link names, which are also different to the link names used for the routines used in C. (End of advice to implementors.)

Advice to users. Callbacks, including the predefined Fortran functions MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and MPI_COMM_NULL_DELETE_FN should not be passed from one application routine that uses the mpi_f08 module to another application routine that uses the mpi module or mpif.h, and vice versa; see also the advice to users on page 688. (End of advice to users.)

MPI_COMM_FREE_KEYVAL(comm_keyval)			15
	、 -	,	16
INOUT	comm_keyval	key value (integer)	17
			18
int MPI_Co	omm_free_keyval(int *comm	n_keyval)	19
MPI Comm	<pre>free_keyval(comm_keyval,</pre>	ierror)	20
INTEGER, INTENT(INOUT) :: comm_keyval			21
	ER, OPTIONAL, INTENT(OUT)		22
			23
	FREE_KEYVAL(COMM_KEYVAL,	IERROR)	24 25
INTEG	ER COMM_KEYVAL, IERROR		23 26
Frees a	an extant attribute key. Thi	s function sets the value of keyval to	20
MPI_KEYVA	AL_INVALID. Note that it is no	t erroneous to free an attribute key that is in use,	28
because the	e actual free does not transpir	e until after all references (in other communicators	29
on the proc	ess) to the key have been freed	l. These references need to be explicitly freed by the	30
program, e	ither via calls to MPI_COMM	_DELETE_ATTR that free one attribute instance,	31
or by calls	to MPI_COMM_FREE that fr	ee all attribute instances associated with the freed	32
communica	communicator.		
	M_SET_ATTR(comm, comm_l	kowal attribute val)	35
		(eyval, attribute_val)	36
INOUT	comm	communicator from which attribute will be attached	37
		(handle)	38
IN	comm_keyval	key value (integer)	39
IN	attribute_val	attribute value	40
			41
int MDT C	omm got ottr(MDI Comm con	m, int comm_keyval, void *attribute_val)	42
IIIC MFI_CO	omm_set_atti(MF1_comm com	m, int comm_keyvar, void *attribute_var)	43
MPI_Comm_	<pre>set_attr(comm, comm_keyva</pre>	al, attribute_val, ierror)	44
TYPE(1	MPI_Comm), INTENT(IN) ::	comm	45
INTEG	ER, INTENT(IN) :: comm_k	xeyval	46
	· · · · · · · · · · · ·		

Unofficial Draft for Comment Only

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

 $\mathbf{5}$

 48

```
1
     MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
\mathbf{2}
          INTEGER COMM, COMM_KEYVAL, IERROR
3
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
4
          This function stores the stipulated attribute value attribute_val for subsequent retrieval
5
      by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if
6
      MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback
7
      function comm_delete_attr_fn was executed), and a new value was next stored. The call
8
      is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an
9
      erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error
10
      code other than MPI_SUCCESS.
11
12
13
      MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)
14
       IN
                 comm
                                              communicator to which the attribute is attached (han-
15
                                              dle)
16
       IN
                 comm_keyval
                                              key value (integer)
17
18
        OUT
                 attribute_val
                                              attribute value, unless flag = false
19
       OUT
                 flag
                                              false if no attribute is associated with the key (logical)
20
21
      int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
22
                     int *flag)
23
^{24}
      MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          INTEGER, INTENT(IN) :: comm_keyval
27
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
28
          LOGICAL, INTENT(OUT) :: flag
29
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
      MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
^{31}
          INTEGER COMM, COMM_KEYVAL, IERROR
32
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
33
34
          LOGICAL FLAG
35
          Retrieves attribute value by key. The call is erroneous if there is no key with value
36
      keyval. On the other hand, the call is correct if the key value exists, but no attribute is
37
      attached on comm for that key; in such case, the call returns flag = false. In particular
38
      MPI_KEYVAL_INVALID is an erroneous key value.
39
40
           Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of
41
           the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the
42
           location where the attribute value is to be returned. Thus, if the attribute value itself is
43
           a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr
44
           will be of type void* and the actual attribute_val parameter to MPI_Comm_get_attr
45
           will be of type void**. (End of advice to users.)
46
47
           Rationale.
                        The use of a formal parameter attribute_val of type void* (rather than
48
           void**) avoids the messy type casting that would be needed if the attribute value is
```

Unofficial Draft for Comment Only

declared with a type other than void*. (End of rationale.) 1			
			2 3
			4
MPI_COM	M_DELETE_ATTR(comm, con	mm_keyval)	5
INOUT	comm	communicator from which the attribute is deleted (han-dle)	6 7
IN	comm_keyval	key value (integer)	8 9
int MPI_C	Comm_delete_attr(MPI_Comm	comm, int comm_keyval)	10 11
MPI_Comm_	_delete_attr(comm, comm_k	eyval, ierror)	12
	(MPI_Comm), INTENT(IN) ::	•	13
INTEC	ER, INTENT(IN) :: comm_1	keyval	14
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	15
MPI_COMM_	DELETE_ATTR(COMM, COMM_K	EYVAL, IERROR)	16 17
	ER COMM, COMM_KEYVAL, IE	-	17
Delete	e attribute from cache by key	This function invokes the attribute delete function	19
		keyval was created. The call will fail if the	20
	-	error code other than MPI_SUCCESS.	21
When	ever a communicator is replic	ated using the function MPI_COMM_DUP or	22
		functions for attributes that are currently set are	23
		a communicator is deleted using the function	24 25
	M_FREE all callback delete f	functions for attributes that are currently set are	25 26
invoked.			27
6.7.3 Wi	ndows		28
0.7.5 101	ndows		29
The functi	ons for caching on windows a	re:	30
			31
MPI_WIN	_CREATE_KEYVAL(win_copy_	.attr_fn, win_delete_attr_fn, win_keyval, extra_state)	32
		· , , , , , , , , , , , , , , , , , , ,	33 34
IN	win_copy_attr_fn	copy callback function for win_keyval (function)	35
IN	win_delete_attr_fn	delete callback function for win_keyval (function)	36
OUT	win_keyval	key value for future access (integer)	37
IN	-	extra state for callback functions	38
IIN	extra_state	extra state for candack functions	39 40
int MPT 4	lin create kevval(MPT Win	_copy_attr_function *win_copy_attr_fn,	40
1	•	function *win_delete_attr_fn,	42
	int *win_keyval, voi	-	43
MDT Win	reate keuval (win convert	tr_fn, win_delete_attr_fn, win_keyval,	44
ти т_wтп_(extra_state, ierror)	n, win_uerete_attr_in, win_Keyval,	45
PROCEDURE(MPI Win conv attr function) ·· win conv attr fn			
		_function) :: win_delete_attr_fn	47 48
40			

```
1
         INTEGER, INTENT(OUT) :: win_keyval
\mathbf{2}
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
5
                   EXTRA_STATE, IERROR)
6
         EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
7
         INTEGER WIN_KEYVAL, IERROR
8
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
9
10
         The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or
11
     MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function
12
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is
13
     a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
14
     attribute_val_out, and returns MPI_SUCCESS.
15
         The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN
16
     from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,
17
     other than returning MPI_SUCCESS.
^{18}
     The C callback functions are:
19
     typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
20
                   void *extra_state, void *attribute_val_in,
21
                   void *attribute_val_out, int *flag);
22
     and
23
     typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
24
                   void *attribute_val, void *extra_state);
25
26
     With the mpi_f08 module, the Fortran callback functions are:
27
     ABSTRACT INTERFACE
28
       SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
^{29}
       attribute_val_in, attribute_val_out, flag, ierror)
30
           TYPE(MPI_Win) :: oldwin
31
           INTEGER :: win_keyval, ierror
32
           INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
33
           attribute_val_out
34
           LOGICAL :: flag
35
     and
36
     ABSTRACT INTERFACE
37
       SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
38
       extra_state, ierror)
39
           TYPE(MPI_Win) :: win
40
           INTEGER :: win_keyval, ierror
41
           INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
42
43
     With the mpi module and mpif.h, the Fortran callback functions are:
^{44}
     SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
45
                   ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
46
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
47
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
48
```

LOGI	ATTRIBUTE_VAL_OUT CAL FLAG		1 2
and			3
	NE WIN DELETE ATTR FUNCTI	ON(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,	4
DODIGOTI	EXTRA_STATE, IERROR)		5
INTE	GER WIN, WIN_KEYVAL, IERR		6 7
) ATTRIBUTE_VAL, EXTRA_STATE	8
If on	attribute conv function or at	tribute delete function returns other than	9
		it to be invoked (for example, MPI_WIN_FREE), is	10
erroneous.			11
01101100 0.01			12
			13
MPI_WIN	_FREE_KEYVAL(win_keyval)		14
INOUT	win_keyval	key value (integer)	15
			16
int MPI_N	/in_free_keyval(int *win_	keyval)	17
MPT Win	free_keyval(win_keyval, i	error)	18 19
	GER, INTENT(INOUT) :: wi		20
	GER, OPTIONAL, INTENT(OUT		21
			22
	FREE_KEYVAL(WIN_KEYVAL, I	ERRUR)	23
INTE	GER WIN_KEYVAL, IERROR		24
			25
	_SET_ATTR(win, win_keyval,	attribute val)	26
	`` <u></u>	,	27
INOUT	win	window to which attribute will be attached (handle)	28 29
IN	win_keyval	key value (integer)	30
IN	attribute_val	attribute value	31
			32
int MPI_N	<pre>/in_set_attr(MPI_Win win,</pre>	<pre>int win_keyval, void *attribute_val)</pre>	33
MPT Win s	set_attr(win, win_keyval,	attribute val jerror)	34
	(MPI_Win), INTENT(IN) ::		35
	GER, INTENT(IN) :: win_k		36
), INTENT(IN) :: attribute_val	37
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	38
MDT LITN (SET_ATTR(WIN, WIN_KEYVAL,		39 40
	GER WIN, WIN_KEYVAL, IERR	-	40 41
	GER (KIND=MPI_ADDRESS_KIND		41
11111	APPOUND AND AND APPOUND ATIND	/ ATTADUTE_VAL	43
			44
			45
			46
			47
			48

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

```
1
     MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)
\mathbf{2}
       IN
                 win
                                             window to which the attribute is attached (handle)
3
                 win_keyval
       IN
                                            key value (integer)
4
5
                 attribute_val
       OUT
                                            attribute value, unless flag = false
6
       OUT
                 flag
                                             false if no attribute is associated with the key (logical)
7
8
     int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
9
                    int *flag)
10
11
     MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
12
          TYPE(MPI_Win), INTENT(IN) :: win
13
          INTEGER, INTENT(IN) :: win_keyval
14
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
15
          LOGICAL, INTENT(OUT) :: flag
16
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
18
          INTEGER WIN, WIN_KEYVAL, IERROR
19
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
20
          LOGICAL FLAG
21
22
23
     MPI_WIN_DELETE_ATTR(win, win_keyval)
24
25
       INOUT
                                             window from which the attribute is deleted (handle)
                 win
26
       IN
                 win_keyval
                                             key value (integer)
27
28
     int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
29
30
     MPI_Win_delete_attr(win, win_keyval, ierror)
^{31}
          TYPE(MPI_Win), INTENT(IN) :: win
32
          INTEGER, INTENT(IN) :: win_keyval
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
35
          INTEGER WIN, WIN_KEYVAL, IERROR
36
37
38
     6.7.4 Datatypes
39
40
     The new functions for caching on datatypes are:
41
42
43
44
45
46
47
48
```

MPI_TYPE	_CREATE_KEYVAL(type_copy extra_state)	_attr_fn, type_delete_attr_fn, type_keyval,	1 2
IN	type_copy_attr_fn	copy callback function for type_keyval (function)	3 4
IN	type_delete_attr_fn	delete callback function for type_keyval (function)	4 5
OUT	type_keyval	key value for future access (integer)	6
IN	extra_state	extra state for callback functions	7 8
int MPI_Ty	• • • • • • •	e_copy_attr_function *type_copy_attr_fn, function *type_delete_attr_fn, d *extra_state)	9 10 11 12
MPI_Type_0	create_keyval(type_copy_a extra_state, ierror)	ttr_fn, type_delete_attr_fn, type_keyval,	13 14
PROCEI INTEGI INTEGI	DURE(MPI_Type_copy_attr_f DURE(MPI_Type_delete_attr ER, INTENT(OUT) :: type_	, INTENT(IN) :: extra_state	15 16 17 18 19 20
EXTERI INTEGI	CREATE_KEYVAL(TYPE_COPY_A EXTRA_STATE, IERROR) NAL TYPE_COPY_ATTR_FN, TY ER TYPE_KEYVAL, IERROR ER(KIND=MPI_ADDRESS_KIND)		20 21 22 23 24 25
MPI_TYPE that does n is a simple-	_DUP_FN from either C or Fo	y be specified as MPI_TYPE_NULL_COPY_FN or ortran. MPI_TYPE_NULL_COPY_FN is a function flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN as flag = 1, returns the value of attribute_val_in in ESS.	26 27 28 29 30
from either other than		ay be specified as MPI_TYPE_NULL_DELETE_FN JLL_DELETE_FN is a function that does nothing,	31 32 33 34
	nt MPI_Type_copy_attr_fun	ction(MPI_Datatype oldtype, ! *extra_state, void *attribute_val_in, ut, int *flag);	35 36 37 38
and typedef in		unction(MPI_Datatype datatype, *attribute_val, void *extra_state);	39 40 41
ABSTRACT SUBROUT attribut TYPI		nction(oldtype, type_keyval, extra_state, out, flag, ierror) pe	42 43 44 45 46 47 48

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

MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val) ¹			
INOUT	datatype	datatype to which attribute will be attached (handle)	2 3
IN	type_keyval	key value (integer)	4
IN	attribute_val	attribute value	5
			6
int MPI_T		<pre>datatype, int type_keyval,</pre>	7 8
	<pre>void *attribute_val)</pre>		9
		eyval, attribute_val, ierror)	10
	MPI_Datatype), INTENT(IN)		11
	ER, INTENT(IN) :: type_k	eyval , INTENT(IN) :: attribute_val	12
	ER, OPTIONAL, INTENT(OUT)		13 14
			15
	ER DATATYPE, TYPE_KEYVAL,	EYVAL, ATTRIBUTE_VAL, IERROR) IERROR	16
	ER(KIND=MPI_ADDRESS_KIND)		17
			18
			19 20
MPI_TYPE	_GET_ATTR(datatype, type_k	eyval, attribute_val, flag)	21
IN	datatype	data type to which the attribute is attached (handle)	22
IN	type_keyval	key value (integer)	23
OUT	attribute_val	attribute value, unless $flag = false$	24 25
OUT	flag	false if no attribute is associated with the key (logical)	23 26
			27
int MPI_T		datatype, int type_keyval,	28
	<pre>void *attribute_val,</pre>	int *flag)	29
MPI_Type_	get_attr(datatype, type_k	eyval, attribute_val, flag, ierror)	30 31
	MPI_Datatype), INTENT(IN)		32
	ER, INTENT(IN) :: type_k	eyval , INTENT(OUT) :: attribute_val	33
	AL, INTENT(OUT) :: flag	, INIENI(UUI) attribute_var	34
	ER, OPTIONAL, INTENT(OUT)	:: ierror	35 36
MPT TYPE	GET ΑΤΤR.(DΑΤΑΤΥΡΕ ΤΥΡΕ Κ	EYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	36 37
	ER DATATYPE, TYPE_KEYVAL,		38
	ER(KIND=MPI_ADDRESS_KIND)		39
LOGIC	AL FLAG		40
			41 42
	_DELETE_ATTR(datatype, typ		43
	、	- ,	44
INOUT	datatype	datatype from which the attribute is deleted (handle)	45
IN	type_keyval	key value (integer)	46
<pre>47 int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval) 48</pre>			

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

Unofficial Draft for Comment Only

```
&gop_key, (void *)0) ) {
                                                                                 1
                                                                                 2
    /* get the key while assigning its copy and delete callback
                                                                                 3
       behavior. */
    } else
                                                                                 4
        MPI_Abort (comm, 99);
                                                                                 5
                                                                                 6
  }
                                                                                 7
                                                                                 8
  MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
                                                                                 9
  if (foundflag)
                                                                                 10
  { /* This module has executed in this group before.
                                                                                11
       We will use the cached information */
  }
                                                                                12
  else
                                                                                13
                                                                                14
  { /* This is a group that we have not yet cached anything in.
                                                                                15
       We will now do so.
                                                                                16
    */
                                                                                 17
                                                                                18
    /* First, allocate storage for the stuff we want,
       and initialize the reference count */
                                                                                19
                                                                                20
                                                                                21
    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
                                                                                22
                                                                                23
                                                                                24
    gop_stuff -> ref_count = 1;
                                                                                25
                                                                                26
    /* Second, fill in *gop_stuff with whatever we want.
       This part isn't shown here */
                                                                                27
                                                                                28
                                                                                29
    /* Third, store gop_stuff as the attribute value */
                                                                                30
    MPI_Comm_set_attr (comm, gop_key, gop_stuff);
                                                                                31
  }
  /* Then, in any case, use contents of *gop_stuff
                                                                                32
                                                                                33
     to do the global op ... */
                                                                                34
}
                                                                                35
/* The following routine is called by MPI when a group is freed \ast/
                                                                                36
                                                                                37
                                                                                38
int gop_stuff_destructor (MPI_Comm comm, int keyval, void *gop_stuffP,
                                                                                39
                       void *extra)
{
                                                                                40
                                                                                41
  gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
                                                                                42
  if (keyval != gop_key) { /* abort -- programming error */ }
                                                                                43
                                                                                44
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff -> ref_count -= 1;
                                                                                45
                                                                                46
                                                                                 47
  /* If no references remain, then free the storage */
                                                                                48
  if (gop_stuff -> ref_count == 0) {
```

```
1
            free((void *)gop_stuff);
\mathbf{2}
          }
3
          return MPI_SUCCESS;
4
        }
5
6
        /* The following routine is called by MPI when a group is copied */
7
        int gop_stuff_copier (MPI_Comm comm, int keyval, void *extra,
8
         void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
9
        {
10
          gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
11
          gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
12
          if (keyval != gop_key) { /* abort -- programming error */ }
13
14
          /* The new group adds one reference to this gop_stuff */
15
          gop_stuff_in -> ref_count += 1;
16
          *gop_stuff_out = gop_stuff_in;
17
          return MPI_SUCCESS;
18
        }
19
```

```
6.8 Naming Objects
```

20

21 22

23

24

25

26

27 28 29 There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

```
MPI_COMM_SET_NAME (comm, comm_name)
```

```
30
       INOUT
                                            communicator whose identifier is to be set (handle)
                comm
^{31}
       IN
                                            the character string which is remembered as the name
                comm_name
32
                                            (string)
33
34
     int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
35
36
     MPI_Comm_set_name(comm, comm_name, ierror)
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         CHARACTER(LEN=*), INTENT(IN) :: comm_name
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
42
         INTEGER COMM, IERROR
         CHARACTER*(*) COMM_NAME
43
44
         MPI_COMM_SET_NAME allows a user to associate a name string with a communicator.
```

The character string which is passed to MPI_COMM_SET_NAME will be saved inside the MPI library (so it can be freed by the caller immediately after the call, or allocated on the ka

IN

stack). Leading spaces in name are significant but trailing ones are not. MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the name of the communicator as seen in the process which made the MPI_COMM_SET_NAME call. There is no requirement that the same (or any) name be assigned to a communicator in every process where it exists. Advice to users. Since MPI_COMM_SET_NAME is provided to help debug code, it is sensible to give the same name to a communicator in all of the processes where it exists, to avoid confusion. (End of advice to users.) The length of the name which can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C to allow for the null terminator. Attempts to put names longer than this will result in truncation of the name. MPI_MAX_OBJECT_NAME must have a value of at least 64. Advice to users. Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed. (End of advice to users.) Advice to implementations. Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. Implementations which allocate space for the name from the heap should still define MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate space for a string of up to this size when calling MPI_COMM_GET_NAME. (End of advice to implementors.) MPI_COMM_GET_NAME (comm, comm_name, resultlen) communicator whose name is to be returned (handle) comm

OUT	comm_name	the name previously stored on the communicator, or	32
		an empty string if no such name exists (string)	33
OUT	resultlen	length of returned name (integer)	34
			35
int MPT C	comm get name(MPI Comm cor	nm, char *comm_name, int *resultlen)	36
	-		37
	get_name(comm, comm_name		38
	TYPE(MPI_Comm), INTENT(IN) :: comm		
	CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name		
	INTEGER, INTENT(OUT) :: resultlen		
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	42
MPT COMM	GET_NAME(COMM, COMM_NAME)	RESULTLEN TERROR)	43
	ER COMM, RESULTLEN, IERR	· · · · ·	44
	CTER*(*) COMM_NAME		45
OIIMIN			46
MPL (COMM GET NAME returns t	he last name which has previously been associated	47

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and retrieved from any language. The 48

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9 10

11

12

13

1415

16

17

18

19 20

21

22

23

 24

25

30

1 same name will be returned independent of the language used. name should be allocated $\mathbf{2}$ so that it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters. 3 MPI_COMM_GET_NAME returns a copy of the set name in name. 4 In C, a null character is additionally stored at name[resultlen]. The value of resultlen $\mathbf{5}$ cannot be larger than MPI_MAX_OBJECT_NAME-1. In Fortran, name is padded on the 6 right with blank characters. The value of resultlen cannot be larger than $\overline{7}$ MPI_MAX_OBJECT_NAME. 8 If the user has not associated a name with a communicator, or an error occurs, 9 MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C). The 10 three predefined communicators will have predefined names associated with them. Thus, 11the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by 12MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of 13MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system 14may have chosen to give a default name to a communicator does not prevent the user from 15setting a name on the same communicator; doing this removes the old name and assigns 16the new one. 17 *Rationale.* We provide separate functions for setting and getting the name of a com-18 19 municator, rather than simply providing a predefined attribute key for the following reasons: 2021• It is not, in general, possible to store a string as an attribute from Fortran. 22• It is not easy to set up the delete function for a string attribute unless it is known 23to have been allocated from the heap. 24 25• To make the attribute key useful additional code to call strdup is necessary. If 26this is not standardized then users have to write it. This is extra unneeded work 27which we can easily eliminate. 28 • The Fortran binding is not trivial to write (it will depend on details of the 29Fortran compilation system), and will not be portable. Therefore it should be in 30 the library rather than in user code. 3132 (End of rationale.) 33 34 Advice to users. The above definition means that it is safe simply to print the string 35returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was 36 no name. 37 Note that associating a name with a communicator has no effect on the semantics of 38 an MPI program, and will (necessarily) increase the store requirement of the program, 39 since the names must be saved. Therefore there is no requirement that users use these 40 functions to associate names with communicators. However debugging and profiling 41 MPI applications may be made easier if names are associated with communicators, 42since the debugger or profiler should then be able to present information in a less 43 cryptic manner. (End of advice to users.) 4445The following functions are used for setting and getting names of datatypes. The 46constant MPI_MAX_OBJECT_NAME also applies to these names. 4748

MPI_TYP	E_SET_NAME (datatype, type	_name)	1		
INOUT	datatype	datatype whose identifier is to be set (handle)	2		
IN	type_name	the character string which is remembered as the name (string)	3 4 5		
		(501118)	5 6		
int MPI_7	Type_set_name(MPI_Datatype	e datatype, const char *type_name)	7		
• -	_set_name(datatype, type_		8 9		
	(MPI_Datatype), INTENT(IN) ACTER(LEN=*), INTENT(IN)	v -	10 11		
INTEG	GER, OPTIONAL, INTENT(OUT)) :: ierror	11		
	_SET_NAME(DATATYPE, TYPE_)	NAME, IERROR)	13		
	GER DATATYPE, IERROR ACTER*(*) TYPE_NAME		14 15		
OIIRID			16		
			17 18		
	E_GET_NAME (datatype, type		19		
IN	datatype	datatype whose name is to be returned (handle)	20		
OUT	type_name	the name previously stored on the datatype, or a empty string if no such name exists (string)	21 22		
OUT	resultlen	length of returned name (integer)	23 24		
int MDT 7			25		
int MPI	<pre>int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,</pre>				
• -	_get_name(datatype, type_		28		
	(MPI_Datatype), INTENT(IN)) :: datatype NAME), INTENT(OUT) :: type_name	29 30		
		ltlen	31		
INTEG	GER, OPTIONAL, INTENT(OUT)) :: ierror	32		
	_GET_NAME(DATATYPE, TYPE_)		33 34		
	GER DATATYPE, RESULTLEN, 1 ACTER*(*) TYPE_NAME	IERROR	35		
			36 37		
	A predefined datatypes have t WCHAR has the default name	he default names of the datatype name. For exam- of MPI WCHAR.	38		
The f	ollowing functions are used fo	r setting and getting names of windows. The con-	39		
stant MPI_{-}	MAX_OBJECT_NAME also app	blies to these names.	40 41		
			42		
	_SET_NAME (win, win_name)		43		
INOUT	win	window whose identifier is to be set (handle)	44 45		
IN	win_name	the character string which is remembered as the name (string)	46		
		(0011118)	47		

```
1
     int MPI_Win_set_name(MPI_Win win, const char *win_name)
\mathbf{2}
     MPI_Win_set_name(win, win_name, ierror)
3
         TYPE(MPI_Win), INTENT(IN) :: win
4
         CHARACTER(LEN=*), INTENT(IN) :: win_name
5
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
6
\overline{7}
     MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
8
         INTEGER WIN, IERROR
9
         CHARACTER*(*) WIN_NAME
10
11
12
     MPI_WIN_GET_NAME (win, win_name, resultlen)
13
       IN
                                            window whose name is to be returned (handle)
                win
14
15
       OUT
                win_name
                                            the name previously stored on the window, or a empty
16
                                            string if no such name exists (string)
17
       OUT
                 resultlen
                                            length of returned name (integer)
18
19
     int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
20
21
     MPI_Win_get_name(win, win_name, resultlen, ierror)
22
         TYPE(MPI_Win), INTENT(IN) :: win
23
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
^{24}
         INTEGER, INTENT(OUT) :: resultlen
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
27
         INTEGER WIN, RESULTLEN, IERROR
28
         CHARACTER*(*) WIN_NAME
29
30
```

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

```
6.9.1 Basic Statements
```

38 When a caller passes a communicator (that contains a context and group) to a callee, that 39 communicator must be free of side effects throughout execution of the subprogram: there 40should be no active operations on that communicator that might involve the process. This 41 provides one model in which libraries can be written, and work "safely." For libraries 42so designated, the callee has permission to do whatever communication it likes with the 43 communicator, and under the above guarantee knows that no other communications will 44interfere. Since we permit good implementations to create new communicators without 45synchronization (such as by preallocated contexts on communicators), this does not impose 46a significant overhead.

47 48

 31

32 33

34

35 36

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

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;

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 $45 \\ 46$

• messages are always selected by source (no use is made of MPI_ANY_SOURCE).

The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated.

- $11 \\ 12$

- - i

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 [44]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [11, 12].

 24

 31

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

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for 26example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord $(0,0)$:	$\operatorname{rank} 0$
coord $(0,1)$:	rank 1
coord $(1,0)$:	rank 2
coord $(1,1)$:	$\operatorname{rank} 3$

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

7.4 Overview of the Functions

43 MPI supports three topology types: Cartesian, graph, and distributed graph. The 44function MPI_CART_CREATE is used to create Cartesian topologies, the function 45

MPI_GRAPH_CREATE is used to create graph topologies, and the functions 46

MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to cre-47

ate distributed graph topologies. These topology creation functions are collective. As with 48

1

2

3

4 $\mathbf{5}$

> 6 7

> > 8

9

10

11

12

13

14

15

25

27

35 36

37

38

39

40 41

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.

MPI defines functions to query a communicator for topology information. The function 22MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. 23Depending 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 2728the 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 2930 can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and 33 MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. 34The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the

4546nearest neighbors on the topology associated with the communicator. The nonblocking 47variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 48 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and

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

 24

2526

 31

32

35

36 37

38

39

40 41

42

43

```
CHAPTER 7. PROCESS TOPOLOGIES
MPI_INEIGHBOR_ALLTOALLW.
7.5
      Topology Constructors
7.5.1 Cartesian Constructor
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
  IN
            comm_old
                                       input communicator (handle)
  IN
            ndims
                                       number of dimensions of Cartesian grid (integer)
  IN
            dims
                                       integer array of size ndims specifying the number of
                                       processes in each dimension
  IN
            periods
                                       logical array of size ndims specifying whether the grid
                                       is periodic (true) or not (false) in each dimension
  IN
            reorder
                                       ranking may be reordered (true) or not (false) (logical)
  OUT
                                       communicator with new Cartesian topology (handle)
            comm_cart
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
               const int periods[], int reorder, MPI_Comm *comm_cart)
MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
    LOGICAL, INTENT(IN) :: periods(ndims), reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
    MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian
topology information is attached. If reorder = false then the rank of each process in the
new group is identical to its rank in the old group. Otherwise, the function may reorder
the processes (possibly so as to choose a good embedding of the virtual topology onto
the physical machine). If the total size of the Cartesian grid is smaller than the size of
the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to
MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created.
The call is erroneous if it specifies a grid that is larger than the group size or if ndims is
negative.
```

41 4243

44

Cartesian Convenience Function: MPI_DIMS_CREATE 7.5.2

45For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced 46distribution of processes per coordinate direction, depending on the number of processes 47in the group to be balanced and optional constraints that can be specified by the user. 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 3

4 5

> 6 7 8

9 10

11

12

13

1415

16

17

18

19 20

21

22 23

 24

25

26

27

28

29

30

 31

32 33

34

35

36

37

38

39

One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an *n*-dimensional topology.

MPI_DIMS	_CREATE(nnodes,	ndims,	dims)	ĺ
----------	----------	---------	--------	-------	---

IN	nnodes	number of nodes in a grid (integer)	6
IN	ndims	number of Cartesian dimensions (integer)	7
	namis	number of eartestan annensions (meeger)	8
INOUT	dims	integer array of size ndims specifying the number of	9
		nodes in each dimension	10
			11
int MPI_D	ims_create(int nnodes, in	nt ndims, int dims[])	12
			13
	create(nnodes, ndims, dim	•	14
	ER, INTENT(IN) :: nnode:	-	15
INTEG	ER, INTENT(INOUT) :: din	ns(ndims)	16
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	17
MPT DTMS	MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)		
	ER NNODES, NDIMS, DIMS(*)	•	19
	, , ,		

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i; only those entries where dims[i] = 0 are modified by the call.

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a multiple of

$$\prod_{i,dims[i]\neq 0} dims[i].$$

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local. If ndims is zero and nnodes is one, MPI_DIMS_CREATE returns MPI_SUCCESS.

Example 7.1

dims	function call	dims
before call		on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

1

 $\mathbf{2}$

3 4 5

20

21

22

23

 24

25

26

27

28 29

30

 31

32

33 34

	314		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.5.3 Graj	oh Constructor	
4	MPI_GRAP	H_CREATE(comm_old, nnode	s, index, edges, reorder, comm_graph)
5 6	IN	comm_old	input communicator (handle)
7	IN	nnodes	number of nodes in graph (integer)
8	IN	index	array of integers describing node degrees (see below)
9 10	IN	edges	array of integers describing graph edges (see below)
10	IN	reorder	ranking may be reordered (true) or not (false) (logical)
12 13	OUT	comm_graph	communicator with graph topology added (handle)
14 15 16	int MPI_Gr	•	_old, int nnodes, const int index[], nt reorder, MPI_Comm *comm_graph)
17	MPI_Graph_	create(comm_old, nnodes,	index, edges, reorder, comm_graph,
18	-	ierror)	
19	TYPE(MPI_Comm), INTENT(IN) :: comm_old		comm_old
20	<pre>INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)</pre>		
21	LOGICAL, INTENT(IN) :: reorder		
22	TYPE(MPI_Comm), INTENT(OUT) ::		G 1
23	INTEGE	CR, OPTIONAL, INTENT(OUT)	:: ierror
24 25	MPI_GRAPH_	CREATE(COMM_OLD, NNODES,	INDEX, EDGES, REORDER, COMM_GRAPH,
25 26		IERROR)	
20	TNTEGE	COMM OLD NNODES INDE	X(*), EDGES(*), COMM GRAPH, TERROR

INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR LOGICAL REORDER

29MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph 30 topology 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.

37 The three parameters nnodes, index and edges define the graph structure. nnodes is the 38number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry 39of array index stores the total number of neighbors of the first i graph nodes. The lists of 40neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. 41 The array edges is a flattened representation of the edge lists. The total number of entries 42in index is nodes and the total number of entries in edges is equal to the number of graph 43edges.

44The definitions of the arguments nnodes, index, and edges are illustrated with the 45following simple example.

46

27

28

```
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:

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

 24

316

¹ 7.5.4 Distributed Graph Constructor

2 MPI_GRAPH_CREATE requires that each process passes the full (global) communication 3 graph to the call. This limits the scalability of this constructor. With the distributed graph 4 interface, the communication graph is specified in a fully distributed fashion. Each process 5specifies only the part of the communication graph of which it is aware. Typically, this 6 could be the set of processes from which the process will eventually receive or get data. 7 or the set of processes to which the process will send or put data, or some combination of 8 such edges. Two different interfaces can be used to create a distributed graph topology. 9 MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with 10 each process specifying each of its incoming and outgoing (adjacent) edges in the logical 11 communication graph and thus requires minimal communication during creation. 12

¹³ MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that ¹⁴ 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

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights,	
outdegree, destinations, destweights, info, reorder, comm_dist_graph)	

23 24	IN	comm_old	input communicator (handle)
25 26	IN	indegree	size of sources and source weights arrays (non-negative integer)
27 28	IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)
29 30 31	IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)
32 33	IN	outdegree	size of destinations and destweights arrays (non-negative integer)
34 35	IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)
36 37 38	IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
39 40	IN	info	hints on optimization and interpretation of weights (handle)
41 42	IN	reorder	the ranks may be reordered $(true) \text{ or not (false) (logical)}$
43 44 45	OUT	comm_dist_graph	communicator with distributed graph topology (handle)
46 47 48	int MPI_I		C(MPI_Comm comm_old, int indegree, const int sourceweights[], int outdegree,
10		55m55 1m5 5501000[],	

```
1
              const int destinations[], const int destweights[],
                                                                                   \mathbf{2}
              MPI_Info info, 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)
                                                                                   6
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                   7
    INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
                                                                                   8
              destinations(outdegree)
                                                                                   9
    INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
                                                                                   10
    TYPE(MPI_Info), INTENT(IN) ::
                                    info
                                                                                   11
    LOGICAL, INTENT(IN) ::
                             reorder
                                                                                   12
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
                                                                                   15
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
                                                                                   16
              OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
                                                                                   17
              COMM_DIST_GRAPH, IERROR)
                                                                                   18
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
                                                                                   19
              DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH,
                                                                                   20
              IERROR
                                                                                   21
    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 40 remapping strategy and other internal MPI optimizations. For instance, approximate count 41 arguments of later communication calls along specific edges could be used as their edge 42weights. Multiplicity of edges can likewise indicate more intense communication between 43 pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 44standard and is left to the implementation. In C or Fortran, an application can supply 45the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have 46 the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some 47but not all processes of comm_old. If the graph is weighted but indegree or outdegree is 48

Unofficial Draft for Comment Only

22

23

24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

1 zero, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to sourceweights $\mathbf{2}$ or destweights respectively. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are 3 not special weight values; rather they are special values for the total array argument. In 4 Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not $\mathbf{5}$ usable for initialization or assignment). See Section 2.5.4. 6 7 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 8 MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then 9 be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case 10 MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.) 11 12Advice to implementors. It is recommended that MPI_UNWEIGHTED not be imple-13 mented as NULL. (End of advice to implementors.) 1415*Rationale.* To ensure backward compatibility, MPI_UNWEIGHTED may still be imple-16mented as NULL. See Annex B.3. (End of rationale.) 1718 The meaning of the info and reorder arguments is defined in the description of the 19following routine. 202122 MPI_DIST_GRAPH_CREATE(comm_old, n, sources, degrees, destinations, weights, info, 23reorder, comm_dist_graph) 24 IN comm_old input communicator (handle) 2526IN number of source nodes for which this process specifies n 27edges (non-negative integer) 28IN array containing the n source nodes for which this prosources 29cess specifies edges (array of non-negative integers) 30 IN degrees array specifying the number of destinations for each 31 source node in the source node array (array of non-32 negative integers) 33 34IN destinations destination nodes for the source nodes in the source 35 node array (array of non-negative integers) 36 IN weights weights for source to destination edges (array of non-37 negative integers) 38 info IN hints on optimization and interpretation of weights 39 (handle) 40 IN 41 reorder the process may be reordered (true) or not (false) (log-42ical) 43 OUT comm_dist_graph communicator with distributed graph topology added 44 (handle) 4546int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[], 47const int degrees[], const int destinations[], 48

```
const int weights[], MPI_Info info, int reorder,
             MPI_Comm *comm_dist_graph)
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
             info, reorder, comm_dist_graph, ierror)
    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 19 distributed graph topology information is attached. Concretely, each process calls the con-20structor with a set of directed (source, destination) communication edges as described below. 21Every process passes an array of n source nodes in the sources array. For each source node, a 22 non-negative number of destination nodes is specified in the degrees array. The destination 23nodes are stored in the corresponding consecutive segment of the destinations array. More 24 precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 25i-th such edge stored in destinations[degrees[0]+ \dots +degrees[i-1]+i]. The weight of this edge is stored in weights $[degrees[0]+\ldots+degrees[i-1]+i]$. Both the sources and the destinations 27arrays may contain the same node more than once, and the order in which nodes are listed 28 as destinations or sources is not significant. Similarly, different processes may specify edges 29 with the same source and destination nodes. Source and destination nodes must be pro-30 cess ranks of comm_old. Different processes may specify different numbers of source and 31destination nodes, as well as different source to destination edges. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination 34 node in the graph specification) are allowed.

The call creates a new communicator **comm_dist_graph** of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE is collective.

If reorder = false, all processes will have the same rank in comm_dist_graph as in $comm_old$. If reorder = true then the MPI library is free to remap to other processes (of comm_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process 45remapping strategy and other internal MPI optimizations. For instance, approximate count 46 arguments of later communication calls along specific edges could be used as their edge 47weights. Multiplicity of edges can likewise indicate more intense communication between 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

1213

14

15

16

17

18

26

32

33

35

36

37

38

39

40

41

42

43

1pairs of processes. However, the exact meaning of edge weights is not specified by the MPI $\mathbf{2}$ standard and is left to the implementation. In C or Fortran, an application can supply 3 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 4 same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not $\mathbf{5}$ all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY 6 or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and $\overline{7}$ MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the 8 total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects 9 like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 10 In the case of an empty weights array argument passed while Advice to users. 11 constructing a weighted graph, one should not pass NULL because the value of 12MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then 13 be indistinguishable from MPI_UNWEIGHTED to the implementation. 14MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.) 1516Advice to implementors. It is recommended that MPI_UNWEIGHTED not be imple-17 mented as NULL. (End of advice to implementors.) 18 19Rationale. To ensure backward compatibility, MPI_UNWEIGHTED may still be imple-20mented as NULL. See Annex B.3. (End of rationale.) 2122 The meaning of the weights argument can be influenced by the info argument. Info 23arguments can be used to guide the mapping; possible options include minimizing the 24 maximum number of edges between processes on different SMP nodes, or minimizing the 25sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it 26is valid for an MPI implementation not to do any reordering. An MPI implementation may 27specify more info key-value pairs. All processes must specify the same set of key-value info 28pairs. 29Advice to implementors. MPI implementations must document any additionally 30 supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the 31 default creation of the distributed graph topology to the MPI library. 32 33 An implementation does not explicitly need to construct the topology from its dis-34 tributed parts. However, all processes can construct the full topology from the dis-35 tributed specification and use this in a call to MPI_GRAPH_CREATE to create the 36 topology. This may serve as a reference implementation of the functionality, and 37 may be acceptable for small communicators. However, a scalable high-quality im-38 plementation would save the topology graph in a distributed way. (End of advice to 39 *implementors.*) 40 41

Example 7.3 As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

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

Unofficial Draft for Comment Only

42

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 $\mathsf{MPI_UNWEIGHTED}$ 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

 31

```
1
\mathbf{2}
     /* get my communication partners along x dimension */
3
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
4
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
5
6
     /* get my communication partners along y dimension */
\overline{7}
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
     destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
8
9
10
     /* get my communication partners along diagonals */
11
     destinations[4] = P*((y+1))(Q)+(x+1)(P); weights[4] = 1;
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
12
13
     destinations[6] = P*((y+1))(Q)+(P+x-1)(P); weights[6] = 1;
14
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
15
16
     sources[0] = rank;
17
     degrees[0] = 8;
18
     MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
19
                              weights, MPI_INFO_NULL, 1, &comm_dist_graph);
20
21
     7.5.5
            Topology Inquiry Functions
22
     If a topology has been defined with one of the above functions, then the topology information
23
     can be looked up using inquiry functions. They all are local calls.
24
25
26
     MPI_TOPO_TEST(comm, status)
27
       IN
                 comm
                                            communicator (handle)
28
29
       OUT
                                            topology type of communicator comm (state)
                 status
30
^{31}
     int MPI_Topo_test(MPI_Comm comm, int *status)
32
     MPI_Topo_test(comm, status, ierror)
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, INTENT(OUT) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
38
          INTEGER COMM, STATUS, IERROR
39
         The function MPI_TOPO_TEST returns the type of topology that is assigned to a
40
     communicator.
41
         The output value status is one of the following:
42
43
       MPI_GRAPH
                                              graph topology
44
                                              Cartesian topology
       MPI_CART
45
                                              distributed graph topology
       MPI_DIST_GRAPH
46
                                              no topology
       MPI_UNDEFINED
47
48
```

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

MPI_GRAI	PHDIMS_GET(comm, nnodes,	nedges)	1		
IN	comm	communicator for group with graph structure (handle)	2 3		
OUT	nnodes	number of nodes in graph (integer) (same as number	4		
		of processes in the group)	5		
OUT	nedges	number of edges in graph (integer)	6		
			7		
int MPI_C	Graphdims_get(MPI_Comm com	mm, int *nnodes, int *nedges)	8 9		
MPI_Graph	ndims_get(comm, nnodes, n	edges, ierror)	10		
-	(MPI_Comm), INTENT(IN) ::	comm	11		
INTEGER, INTENT(OUT) :: nnodes, nedges					
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	13		
MPI_GRAPH	HDIMS_GET(COMM, NNODES, N	EDGES, IERROR)	14 15		
INTEG	GER COMM, NNODES, NEDGES,	IERROR	16		
Funct	ions MPI_GRAPHDIMS_GET	and MPI_GRAPH_GET retrieve the graph-topology	17		
		communicator by MPI_GRAPH_CREATE.	18		
	- 0	GRAPHDIMS_GET can be used to dimension the	19		
vectors inc	lex and edges correctly for the	e following call to MPI_GRAPH_GET.	20		
			21 22		
MPI_GRAI	PH_GET(comm, maxindex, ma	xedges, index, edges)	23		
IN	comm	communicator with graph structure (handle)	24		
IN	maxindex	length of vector index in the calling program	25		
		(integer)	26 27		
IN	maxedges	length of vector edges in the calling program	27		
		(integer)	29		
OUT	index	array of integers containing the graph structure (for	30		
		details see the definition of $MPI_GRAPH_CREATE)$	31		
OUT	edges	array of integers containing the graph structure	32 33		
			33 34		
int MPI_C		<pre>int maxindex, int maxedges, int index[],</pre>	35		
	<pre>int edges[])</pre>		36		
-		edges, index, edges, ierror)	37		
	(MPI_Comm), INTENT(IN) ::		38		
	ER, INTENT(IN) :: maxin FR INTENT(OUT) ·· inde	dex, maxedges x(maxindex), edges(maxedges)	$\frac{39}{40}$		
	ER, OPTIONAL, INTENT(OUT	o o	41		
			42		
		EDGES, INDEX, EDGES, IERROR) GES, INDEX(*), EDGES(*), IERROR	43		
10100			44		
			$45 \\ 46$		
			47		
			48		

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

1 MPI_CART_RANK(comm, coords, rank) $\mathbf{2}$ IN communicator with Cartesian structure (handle) comm 3 IN coords integer array (of size ndims) specifying the Cartesian 4 coordinates of a process 5 6 OUT rank rank of specified process (integer) 7 int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank) 9 MPI_Cart_rank(comm, coords, rank, ierror) 10 TYPE(MPI_Comm), INTENT(IN) :: comm 11 INTEGER, INTENT(IN) :: coords(*) 12INTEGER, INTENT(OUT) :: rank 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14 15MPI_CART_RANK(COMM, COORDS, RANK, IERROR) 16INTEGER COMM, COORDS(*), RANK, IERROR 17For a process group with Cartesian structure, the function MPI_CART_RANK trans-18 lates the logical process coordinates to process ranks as they are used by the point-to-point 19 routines. 20For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that 21is, coords(i) < 0 or $coords(i) \ge dims(i)$, it is shifted back to the interval 22 $0 \leq coords(i) < dims(i)$ automatically. Out-of-range coordinates are erroneous for non-23periodic dimensions. 24 If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-25icant and 0 is returned in rank. 262728MPI_CART_COORDS(comm, rank, maxdims, coords) 29 IN communicator with Cartesian structure (handle) comm 30 31IN rank of a process within group of comm (integer) rank 32 IN maxdims length of vector **coords** in the calling program (integer) 33 OUT coords integer array (of size ndims) containing the Cartesian 34 coordinates of specified process (array of integers) 35 36 37 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]) 38 MPI_Cart_coords(comm, rank, maxdims, coords, ierror) 39 TYPE(MPI_Comm), INTENT(IN) :: comm 40 INTEGER, INTENT(IN) :: rank, maxdims 41 INTEGER, INTENT(OUT) :: coords(maxdims) 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR) 45INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR 46The inverse mapping, rank-to-coordinates translation is provided by 47MPI_CART_COORDS. 48

Unofficial Draft for Comment Only

1 If comm is associated with a zero-dimensional Cartesian topology, $\mathbf{2}$ coords will be unchanged. 3 4 MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors) 56 IN communicator with graph topology (handle) comm 7 IN rank of process in group of comm (integer) rank 8 OUT nneighbors number of neighbors of specified process (integer) 9 10 11int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors) 12MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) 13 TYPE(MPI_Comm), INTENT(IN) :: comm 14INTEGER, INTENT(IN) :: rank 15INTEGER, INTENT(OUT) :: nneighbors 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR) 19INTEGER COMM, RANK, NNEIGHBORS, IERROR 202122MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors) 23IN communicator with graph topology (handle) comm 24 IN rank rank of process in group of comm (integer) 2526IN maxneighbors size of array neighbors (integer) 27OUT neighbors ranks of processes that are neighbors to specified pro-28cess (array of integer) 29 30 int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, 31 int neighbors[]) 32 33 MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) 34 TYPE(MPI_Comm), INTENT(IN) :: comm 35 INTEGER, INTENT(IN) :: rank, maxneighbors 36 INTEGER, INTENT(OUT) :: neighbors(maxneighbors) 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) 39 INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR 4041 MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency 42information for a graph topology. The returned count and array of neighbors for the queried 43 rank will both include all neighbors and reflect the same edge ordering as was specified by 44 the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT 45 and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array 46 passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1] 47 is zero): 48

7.5. TOPOLOGY CONSTRUCTORS

- The number of neighbors (nneighbors) returned from MPI_GRAPH_NEIGHBORS_COUNT will be (index[rank] index[rank-1]).
- The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-1]] through edges[index[rank]-1].

Example 7.5

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

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

Thus, the input arguments to MPI_GRAPH_CREATE are:

nnodes = 4index = 3, 5, 6, 9edges = 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

Unofficial Draft for Comment Only

7 8

9

18

19

20

21 22

23

32

33

34

35

36

1 2

3

4

Suppose that the communicator **comm** has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
4
     ! assume: each process has stored a real number A.
5
        extract neighborhood information
6
     !
            CALL MPI_COMM_RANK(comm, myrank, ierr)
7
            CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
8
9
     ! perform exchange permutation
            CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
10
11
                 neighbors(1), 0, comm, status, ierr)
     ! perform shuffle permutation
12
            CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
13
                 neighbors(3), 0, comm, status, ierr)
14
     ! perform unshuffle permutation
15
16
            CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &
17
                 neighbors(2), 0, comm, status, ierr)
18
         MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS pro-
19
     vide adjacency information for a distributed graph topology.
20
21
22
     MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)
23
       IN
                comm
                                           communicator with distributed graph topology (han-
^{24}
                                           dle)
25
26
       OUT
                indegree
                                           number of edges into this process (non-negative inte-
27
                                           ger)
28
       OUT
                outdegree
                                           number of edges out of this process (non-negative in-
29
                                           teger)
30
       OUT
                weighted
                                           false if MPI_UNWEIGHTED was supplied during cre-
^{31}
                                           ation, true otherwise (logical)
32
33
34
     int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
35
                    int *outdegree, int *weighted)
36
     MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, INTENT(OUT) :: indegree, outdegree
39
         LOGICAL, INTENT(OUT) :: weighted
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
43
         INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
44
         LOGICAL WEIGHTED
45
46
47
48
```

1

 $\mathbf{2}$

MPI_DIS	C_GRAPH_NEIGHBORS(comm destinations, destweights	, maxindegree, sources, sourceweights, maxoutdegree,)	1 2	
IN	comm	communicator with distributed graph topology (han- dle)	3 4 5	
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)	6 7	
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)	8 9	
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)	10 11 12	
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)	12 13 14	
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	15 16	
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	17 18 19	
<pre>int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>				
<pre>MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: maxindegree, maxoutdegree INTEGER, INTENT(OUT) :: sources(maxindegree), destinations(maxoutdegree) INTEGER :: sourceweights(*), destweights(*) INTEGER, OPTIONAL, INTENT(OUT) :: ierror </pre>				
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR				
These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (poten- tially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by 41				

each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in comm_old in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on

MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for

sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays.

If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for

42

43

44

45 46

47

the order of values in sources and destinations is that two calls to the routine with same in put argument comm will return the same sequence of edges. If maxindegree or maxoutdegree
 is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then
 only the first part of the full list is returned.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

¹³ ¹⁴ If the process topology is a Cartesian structure, an MPI_SENDRECV operation may be used ¹⁵ along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes ¹⁶ the rank of a source process for the receive, and the rank of a destination process for the ¹⁷ send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides ¹⁸ the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. ¹⁹ The user specifies the coordinate direction and the size of the step (positive or negative). ²⁰ The function is local.

```
21
22
```

23

5

6

7

8

9

10 11

12

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

23					
24	IN	comm	communicator with Cartesian structure (handle)		
25	IN	direction	coordinate dimension of shift (integer)		
26 27 28	IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)		
29	OUT	rank_source	rank of source process (integer)		
30	OUT	rank_dest	rank of destination process (integer)		
31					
32	int MPI_Ca	art_shift(MPI_Comm comm,	int direction, int disp,		
$33 \\ 34$		<pre>int *rank_source, in</pre>	t *rank_dest)		
35	MDT Court shift (some dimension dian work source work doct is work)				
36					
37		ER, INTENT(IN) :: direc	-		
38		ER, INTENT(OUT) :: rank	-		
39	INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror		
40 41	MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)				
42	INTEG	ER COMM, DIRECTION, DISP	, RANK_SOURCE, RANK_DEST, IERROR		
43		0	e coordinate dimension to be traversed by the shift.		
44					
45	-		e Cartesian group in the specified coordinate direc-		
46	,	-	entifiers for a circular or an end-off shift. In the case NULL may be returned in rank_source or rank_dest,		
47 48			ation for the shift is out of range.		
40	mananing		tor the shift is out of funge.		

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

Example 7.7

The communicator, **comm**, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of **REALs** is stored one element per process, in variable **A**. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.

! find process rank
CALL MPI_COMM_RANK(comm, rank, ierr)
! find Cartesian coordinates
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
! compute shift source and destination
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
! skew array
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
status, ierr)

Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (*End of advice to users.*)

7.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	comm	communicator with Cartesian structure (handle)	31	
IN	remain_dims	the i-th entry of remain_dims specifies whether the	32 33	
		i-th dimension is kept in the subgrid (true) or is drop-	33 34	
		ped (false) (logical vector)	35	
OUT	newcomm	communicator containing the subgrid that includes	36	
		the calling process (handle)	37	
			38	
int MPI_Ca	<pre>int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)</pre>			
MPT Cart o	MPI_Cart_sub(comm, remain_dims, newcomm, ierror)			
	<pre>IPI_Comm, IEMAIN_dIMS, IN IPI_Comm), INTENT(IN) ::</pre>		41	
	AL, INTENT(IN) :: remain		42	
	IPI_Comm), INTENT(OUT) ::		43	
INTEGE	R, OPTIONAL, INTENT(OUT)) :: ierror	44 45	
MDT CADT C	SUB(COMM, REMAIN_DIMS, NE		46	
	R COMM, NEWCOMM, IERROR	Swoorm, Heitholt/	47	
	AL REMAIN_DIMS(*)		48	

Unofficial Draft for Comment Only

 $\overline{7}$

 24

1 If a Cartesian topology has been created with MPI_CART_CREATE, the function $\mathbf{2}$ MPI_CART_SUB can be used to partition the communicator group into subgroups that 3 form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator 4 with the associated subgrid Cartesian topology. If all entries in remain_dims are false or $\mathbf{5}$ comm is already associated with a zero-dimensional Cartesian topology then newcomm is 6 associated with a zero-dimensional Cartesian topology. (This function is closely related to $\overline{7}$ MPI_COMM_SPLIT.) 8 Example 7.8 9 Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let 10 remain_dims = (true, false, true). Then a call to 11 12MPI_CART_SUB(comm, remain_dims, comm_new); 13 will create three communicators each with eight processes in a 2×4 Cartesian topology. 14If remain_dims = (false, false, true) then the call to $MPI_CART_SUB(comm, remain_dims,$ 15comm_new) will create six non-overlapping communicators, each with four processes, in a 1617one-dimensional Cartesian topology. 18 197.5.8 Low-Level Topology Functions 20The two additional functions introduced in this section can be used to implement all other 21topology functions. In general they will not be called by the user directly, unless he or she 22 is creating additional virtual topology capability other than that provided by MPI. The two 23calls are both local. 242526MPI_CART_MAP(comm, ndims, dims, periods, newrank) 27IN comm input communicator (handle) 28IN number of dimensions of Cartesian structure (integer) ndims 2930 IN dims integer array of size ndims specifying the number of 31 processes in each coordinate direction 32 logical array of size ndims specifying the periodicity IN periods 33 specification in each coordinate direction 34 OUT newrank reordered rank of the calling process; 35 MPI_UNDEFINED if calling process does not belong 36 to grid (integer) 37 38 39 int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[], 40const int periods[], int *newrank) 41 MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) 42TYPE(MPI_Comm), INTENT(IN) :: comm 43 INTEGER, INTENT(IN) :: ndims, dims(ndims) 44 LOGICAL, INTENT(IN) :: periods(ndims) 45INTEGER, INTENT(OUT) :: newrank 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)

INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR 1 $\mathbf{2}$ LOGICAL PERIODS(*) 3 MPI_CART_MAP computes an "optimal" placement for the calling process on the phys-4 ical machine. A possible implementation of this function is to always return the rank of the 5 calling process, that is, not to perform any reordering. 6 7 The function MPI_CART_CREATE(comm, ndims, dims, Advice to implementors. 8 periods, reorder, comm_cart), with reorder = true can be implemented by calling 9 MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling 10 MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank \neq 11 MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims 12is zero then a zero-dimensional Cartesian topology is created. 13 The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented 14 by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number 15encoding of the lost dimensions as color and a single number encoding of the preserved 16 dimensions as key. 17 18 All other Cartesian topology functions can be implemented locally, using the topology 19information that is cached with the communicator. (End of advice to implementors.) 20The corresponding function for graph structures is as follows. 2122 23MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) 24 25IN comm input communicator (handle) 26IN nnodes number of graph nodes (integer) 27IN index integer array specifying the graph structure, see 28MPI_GRAPH_CREATE 29 30 IN edges integer array specifying the graph structure 31OUT newrank reordered rank of the calling process; 32 MPI_UNDEFINED if the calling process does not be-33 long to graph (integer) 34 35 int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 36 const int edges[], int *newrank) 37 38 MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) 39 TYPE(MPI_Comm), INTENT(IN) :: comm 40 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*) 41 INTEGER, INTENT(OUT) :: newrank 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 44INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 454647Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index,

edges, reorder, comm_graph), with reorder = true can be implemented by calling

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

7.6 Neighborhood Collective Communication on Process Topologies

¹⁰ MPI process topologies specify a communication graph, but they implement no commu-¹¹ nication function themselves. Many applications require sparse nearest neighbor commu-¹² nications that can be expressed as graph topologies. We now describe several collective ¹³ operations that perform communication along the edges of a process topology. All of these ¹⁴ functions are collective; i.e., they must be called by all processes in the specified commu-¹⁵ nicator. See Section 5 for an overview of other dense (global) collective communication ¹⁶ operations and the semantics of collective operations.

¹⁶ operations and the semantics of conective operations. ¹⁷ If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources ¹⁸ and destinations containing 0, ..., n-1, where n is the number of processes in the group ¹⁹ of comm_old (i.e., the graph is fully connected and also includes an edge from each node ²⁰ to itself), then the sparse neighborhood communication routine performs the same data ²¹ exchange as the corresponding dense (fully-connected) collective operation. In the case of a ²² Cartesian communicator, only nearest neighbor communication is provided, corresponding ²³ to rank_source and rank_dest in MPI_CART_SHIFT with input disp=1.

23 24

25

26

27

28

29

30

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.)

 31 For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-32 quence of neighbors in the send and receive buffers at each process is defined as the sequence 33 returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For 34a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood col-35 lective communication is restricted to adjacency matrices, where the number of edges be-36 tween any two processes is defined to be the same for both processes (i.e., with a symmetric 37 adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 38 defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that 39 general graph topologies should generally be replaced by the distributed graph topologies.

⁴⁰ For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neigh-⁴¹ bors in the send and receive buffers at each process is defined by order of the dimensions, ⁴² first the neighbor in the negative direction and then in the positive direction with dis-⁴³ placement 1. The numbers of sources and destinations in the communication routines are ⁴⁴ 2*ndims with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at ⁴⁵ the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., ⁴⁶ periods[...]==false), then this neighbor is defined to be MPI_PROC_NULL.

⁴⁷ If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collec-⁴⁸ tive communication behaves like a point-to-point communication with MPI_PROC_NULL in

Unofficial Draft for Comment Only

1

2

3

4

5

6 7

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.

MPI NEIGH	HBOR ALLGATHER(sendbuf.	sendcount, sendtype, recvbuf, recvcount, recvtype,	11
_	comm)		12
IN	sendbuf	starting address of send buffer (choice)	13 14
IN	sendcount	Ű ()	15
IIN	Senacount	number of elements sent to each neighbor (non-negative integer)	16
		<i>,</i>	17
IN	sendtype	data type of send buffer elements (handle)	18
OUT	recvbuf	starting address of receive buffer (choice)	19
IN	recvcount	number of elements received from each neighbor (non-	20
		negative integer)	21
IN	recvtype	data type of receive buffer elements (handle)	22
IN	comm	communicator with topology structure (handle)	23 24
IIN	IN comm communicator with topology structure (handle)		
· · NDT N			25
int MPI_Ne	0 0	roid* sendbuf, int sendcount,	26
	VI VI	e, void* recvbuf, int recvcount,	27
	MPI_Datatype recvtype	e, MPI_Comm comm)	28
MPI_Neighl	bor_allgather(sendbuf, se	ndcount, sendtype, recvbuf, recvcount,	29 30
	recvtype, comm, ierro	or)	31
TYPE(>	*), DIMENSION(), INTENT	'(IN) :: sendbuf	32
TYPE(>	TYPE(*), DIMENSION() :: recvbuf		
INTEG	INTEGER, INTENT(IN) :: sendcount, recvcount		
TYPE(1	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype		
TYPE(1	TYPE(MPI_Comm), INTENT(IN) :: comm		
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 38			

RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If **comm** is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);

1 2

3 4

5

6

7

8

9 10

39

40

 $41 \\ 42$

43

44

45

 $46 \\ 47$

```
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
7
      /* assume sendbuf and recvbuf are of type (char*) */
8
      for(k=0; k<outdegree; ++k)</pre>
9
        MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
10
11
      for(l=0; l<indegree; ++1)</pre>
12
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
13
                    srcs[1]...);
14
15
     MPI_Waitall(...);
16
          Figure 7.1 shows the neighborhood gather communication of one process with outgoing
17
      neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf to
18
      all four destinations (outgoing neighbors) and it will receive the contribution from all six
19
      sources (incoming neighbors) into separate locations of its receive buffer.
20
21
                                              d_0
22
                                                              d_2, s_4
23
                                           s_0
24
25
                               d_1
                                                          s_1
26
27
28
                                                                  s_3
                                           s_2
29
30
                                                      d_{3}, s_{5}
^{31}
                       sendbuf
32
33
34
                                                                   s_4
                                                                           s_5
                                    s_0
                                           s_1
                                                   s_2
                                                           s_3
35
                       recvbuf
36
37
                      Figure 7.1: Neighborhood gather communication example.
38
39
          All arguments are significant on all processes and the argument comm must have iden-
40
      tical values on all processes.
41
          The type signature associated with sendcount, sendtype, at a process must be equal to
42
      the type signature associated with recvcount, recvtype at all other processes. This implies
43
      that the amount of data sent must be equal to the amount of data received, pairwise between
44
      every pair of communicating processes. Distinct type maps between sender and receiver are
45
      still allowed.
46
```

47 Rationale. For optimization reasons, the same type signature is required indepen 48 dently of whether the topology graph is connected or not. (*End of rationale.*)

7.6. NEIGHBORHOOD COLLECTIVE COMMUNICATION

1 The "in place" option is not meaningful for this operation. The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different 2 3 numbers of elements from each neighbor. 4 5MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 6 recvtype, comm) 7 8 IN sendbuf starting address of send buffer (choice) 9 IN sendcount number of elements sent to each neighbor (non-negative 10 integer) 11 IN sendtype data type of send buffer elements (handle) 1213 OUT recvbuf starting address of receive buffer (choice) 14IN recvcounts non-negative integer array (of length indegree) con-15taining the number of elements that are received from 16 each neighbor 17IN displs integer array (of length indegree). Entry i specifies the 18 displacement (relative to recvbuf) at which to place the 19 incoming data from neighbor i 2021IN data type of receive buffer elements (handle) recvtype 22 IN communicator with topology structure (handle) comm 23 24 int MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount, 25MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 26const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 2728 MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 29 displs, recvtype, comm, ierror) 30 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 31TYPE(*), DIMENSION(..) :: recvbuf 32 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*) 33 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(MPI_Comm), INTENT(IN) :: comm 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 37 DISPLS, RECVTYPE, COMM, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 40 IERROR 41

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted); int *srcs=(int*)malloc(indegree*sizeof(int));

Unofficial Draft for Comment Only

42

43

44

45

 $46 \\ 47$

```
1
     int *dsts=(int*)malloc(outdegree*sizeof(int));
\mathbf{2}
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
3
                                  outdegree,dsts,MPI_UNWEIGHTED);
4
     int k,l;
5
6
     /* assume sendbuf and recvbuf are of type (char*) */
\overline{7}
     for(k=0; k<outdegree; ++k)</pre>
8
       MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
9
10
     for(l=0; l<indegree; ++l)</pre>
11
       MPI_Irecv(recvbuf+displs[l]*extent(recvtype), recvcounts[l], recvtype,
12
                   srcs[1],...);
13
14
     MPI_Waitall(...);
```

 $14 \\ 15$

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[I], recvtype at any other process with srcs[I]==j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data received from the I-th neighbor is placed into recvbuf beginning at offset displs[I] elements (in terms of the recvtype).

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

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

25 26

27

34

22

7.6.2 Neighbor Alltoall

In this function, each process i receives data items from each process j if an edge (j,i)exists in the topology graph or Cartesian topology. Similarly, each process i sends data items to all processes j where an edge (i, j) exists. This call is more general than

³¹ MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. ³² The *k*-th block in send buffer is sent to the *k*-th neighboring process and the *l*-th block in ³³ the receive buffer is received from the *l*-th neighbor.

- 35

 36

 37

 38

 39

 40

 41

 42

 43

 44

 45

 46
- 47 48

MPI_	NEIGHBOR_ALLTOALL(se comm)	endbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative	4 5
	Schacount	integer)	6
IN	sendtype	data type of send buffer elements (handle)	7
OU	Г recvbuf	starting address of receive buffer (choice)	8 9
IN	recvcount	number of elements received from each neighbor (non-negative integer)	9 10 11
IN	recvtype	data type of receive buffer elements (handle)	12
IN	comm	communicator with topology structure (handle)	13 14
int M	PI_Neighbor_alltoall(const void* sendbuf, int sendcount,	15 16
	-	sendtype, void* recvbuf, int recvcount,	10
	MPI_Datatype	recvtype, MPI_Comm comm)	18
MPI_N	leighbor_alltoall(send	buf, sendcount, sendtype, recvbuf, recvcount,	19
	recvtype, com		20
		, INTENT(IN) :: sendbuf	21 22
	<pre>YPE(*), DIMENSION() </pre>		22
		sendcount, recvcount	24
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm		
	INTEGER, OPTIONAL, INT		26
MPT N	IETGHBOR ALLTOALL(SEND)	BUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	27
· · · · ·	RECVTYPE, COM		28 29
<	<type> SENDBUF(*), RECVBUF(*)</type>		
I	INTEGER SENDCOUNT, SEN	DTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	30 31
Г	This function supports Cart	tesian communicators, graph communicators, and distributed	32
		bed in Section 7.6. If comm is a distributed graph commu-	33
· ·		ch process executed sends to each of its outgoing neighbors	34
and re	eceives from each of its ine	coming neighbors:	35
			36
		<pre>ount(comm,&indegree,&outdegree,&weighted);</pre>	37 38
	<pre>int *srcs=(int*)malloc(indegree*sizeof(int)); int *dsts=(int*)malloc(outdegree*sizeof(int));</pre>		
		omm,indegree,srcs,MPI_UNWEIGHTED,	40
		utdegree,dsts,MPI_UNWEIGHTED);	41
int k		5	42
			43
		buf are of type (char*) */	44
	x=0; k <outdegree; ++k)<="" td=""><td>count tort (academa) condecunt condema</td><td>45 46</td></outdegree;>	count tort (academa) condecunt condema	45 46
MPI	Isend(sendbuf+k*send dsts[k],);	count*extent(sendtype),sendcount,sendtype,	40
	usus[k],/,		48

```
1
     for(l=0; l<indegree; ++1)</pre>
\mathbf{2}
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
3
                    srcs[1],...);
4
5
     MPI_Waitall(...);
6
          The type signature associated with sendcount, sendtype, at a process must be equal to
7
      the type signature associated with recvcount, recvtype at any other process. This implies
8
      that the amount of data sent must be equal to the amount of data received, pairwise between
9
      every pair of communicating processes. Distinct type maps between sender and receiver are
10
     still allowed.
11
          The "in place" option is not meaningful for this operation.
12
          All arguments are significant on all processes and the argument comm must have iden-
13
      tical values on all processes.
14
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
15
      numbers of elements to and from each neighbor.
16
17
18
      MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
19
                      rdispls, recvtype, comm)
20
        IN
                  sendbuf
                                                starting address of send buffer (choice)
21
22
        IN
                  sendcounts
                                                non-negative integer array (of length outdegree) speci-
23
                                                fying the number of elements to send to each neighbor
^{24}
        IN
                  sdispls
                                                integer array (of length outdegree). Entry j specifies
25
                                                the displacement (relative to sendbuf) from which to
26
                                                send the outgoing data to neighbor j
27
        IN
                  sendtype
                                                data type of send buffer elements (handle)
28
29
        OUT
                  recvbuf
                                                starting address of receive buffer (choice)
30
        IN
                  recvcounts
                                                non-negative integer array (of length indegree) speci-
^{31}
                                                fying the number of elements that are received from
32
                                                each neighbor
33
        IN
                  rdispls
                                                integer array (of length indegree). Entry i specifies the
34
                                                displacement (relative to recvbuf) at which to place the
35
                                                incoming data from neighbor i
36
37
        IN
                                                data type of receive buffer elements (handle)
                  recvtype
38
        IN
                  comm
                                                communicator with topology structure (handle)
39
40
      int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],
41
                      const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
42
                      const int recvcounts[], const int rdispls[],
43
                      MPI_Datatype recvtype, MPI_Comm comm)
44
45
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
46
                      recvcounts, rdispls, recvtype, comm, ierror)
47
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
48
          TYPE(*), DIMENSION(..) :: recvbuf
```

```
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
              rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::
                                  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                       ierror
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. 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:

<pre>MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);</pre>		
<pre>int *srcs=(int*)malloc(indegree*sizeof(int));</pre>		
<pre>int *dsts=(int*)malloc(outdegree*sizeof(int));</pre>		
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,		
<pre>outdegree,dsts,MPI_UNWEIGHTED);</pre>		

int k,l;

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)</pre>
```

```
MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
          dsts[k],...);
```

```
for(l=0; l<indegree; ++1)</pre>
 MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype),recvcounts[1],recvtype,
            srcs[1],...);
```

```
MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process 35i must be equal to the type signature associated with recvcounts[I], recvtype with srcs[I]==i 36 at process j. This implies that the amount of data sent must be equal to the amount of 37 data received, pairwise between every pair of communicating processes. Distinct type maps 38 between sender and receiver are still allowed. The data in the sendbuf beginning at offset 39 sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data received from the I-th incoming neighbor is placed into recvbuf beginning at offset rdispls[I] 41 elements (in terms of the recvtype). 42

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

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

MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

1213

14

15

1617

22 23

 24

25

26

2728

29

30

3132

33 34

40

43

44

45

46

MPI_NEI	GHBOR_ALLTOALLW(sendbuf rdispls, recvtypes, comm	, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,	
IN	sendbuf	starting address of send buffer (choice)	
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 the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)	
IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)	
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 in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)	
IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)	
IN	comm	communicator with topology structure (handle)	
int MPI_Neighbor_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)			
TYPE TYPE INTE INTE TYPE TYPE INTE MPI_NEIG <type INTE</type 	recvcounts, rdispls (*), DIMENSION(), INTEN (*), DIMENSION() :: r GER, INTENT(IN) :: sendo GER(KIND=MPI_ADDRESS_KINI (MPI_Datatype), INTENT(IN (MPI_Comm), INTENT(IN) :: GER, OPTIONAL, INTENT(OUT HBOR_ALLTOALLW(SENDBUF, S RECVCOUNTS, RDISPLS e> SENDBUF(*), RECVBUF(*) GER(KIND=MPI_ADDRESS_KINI	recvbuf counts(*), recvcounts(*) D), INTENT(IN) :: sdispls(*), rdispls(*) N) :: sendtypes(*), recvtypes(*) : comm F) :: ierror SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, , RECVTYPES, COMM, IERROR) D) SDISPLS(*), RDISPLS(*)	
	IN IN IN IN OUT IN IN IN IN IN IN IN MPI_Neig MPI_Neig TYPE INTE INTE INTE INTE INTE	rdispls, recvtypes, comm IN sendbuf IN sendcounts IN sdispls IN sdispls IN sendtypes OUT recvbuf IN recvcounts IN rdispls IN rdispls IN recvtypes IN comm int MPI_Neighbor_alltoallw(const const MPI_Aint sdis void* recvbuf, cons const MPI_Aint rdis MPI_Comm comm) MPI_Neighbor_alltoallw(sendbuf, s recvcounts, rdispls TYPE(*), DIMENSION(), INTEN TYPE(*), DIMENSION() :: n INTEGER, INTENT(IN) :: sendo INTEGER, INTENT(IN) :: sendo INTEGER, OPTIONAL, INTENT(OUT) MPI_NEIGHBOR_ALLTOALLW(SENDBUF, S	

IERROR

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
outdegree,dsts,MPI_UNWEIGHTED);
```

int k,l;

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
   MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);</pre>
```

```
for(1=0; 1<indegree; ++1)
    MPI_Irecv(recvbuf+rdispls[1],recvcounts[1], recvtypes[1],srcs[1],...);</pre>
```

```
MPI_Waitall(...);
```

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

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

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

7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 5.12. 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

	344		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.7.1 Nor	nblocking Neighborhood Gath	ier
4 5	MPI_INEIG	HBOR_ALLGATHER(sendbuf, comm, request)	sendcount, sendtype, recvbuf, recvcount, recvtype,
6 7	IN	sendbuf	starting address of send buffer (choice)
8 9	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
10	IN	sendtype	data type of send buffer elements (handle)
11 12	OUT	recvbuf	starting address of receive buffer (choice)
13 14	IN	recvcount	number of elements received from each neighbor (non-negative integer)
15	IN	recvtype	data type of receive buffer elements (handle)
16 17	IN	comm	communicator with topology structure (handle)
18	OUT	request	communication request (handle)
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	TYPE(TYPE(INTEG TYPE(TYPE(TYPE(INTEG MPI_INEIG (type INTEG	<pre>MPI_Datatype recvtype hbor_allgather(sendbuf, s recvtype, comm, reque *), DIMENSION(), INTENT *), DIMENSION(), ASYNCH ER, INTENT(IN) :: sendco MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) ER, OPTIONAL, INTENT(OUT) HBOR_ALLGATHER(SENDBUF, S RECVTYPE, COMM, REQUE > SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, R</pre>	<pre>(IN), ASYNCHRONOUS :: sendbuf RONOUS :: recvbuf ount, recvcount :: sendtype, recvtype comm :: request :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,</pre>

MPI <u>.</u>	INEIGHBOR_ALLGATHERV(s_ recvtype, comm, re	sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, quest)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
οι	JT recvbuf	starting address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor	9 10 11 12
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	13 14 15
IN	recvtype	data type of receive buffer elements (handle)	16 17
IN	comm	communicator with topology structure (handle)	18
οι	JT request	communication request (handle)	19
			20
int	MPI_Datatype ser	<pre>(const void* sendbuf, int sendcount, ndtype, void* recvbuf, const int recvcounts[], s[], MPI_Datatype recvtype, MPI_Comm comm, quest)</pre>	21 22 23 24 25
	displs, recvtype TYPE(*), DIMENSION(), I TYPE(*), DIMENSION(), A INTEGER, INTENT(IN) :: s INTEGER, INTENT(IN), ASYM	sendcount NCHRONOUS :: recvcounts(*), displs(*) NT(IN) :: sendtype, recvtype N) :: comm C(OUT) :: request	26 27 28 29 30 31 32 33 34 35
	MPI_INEIGHBOR_ALLGATHERV (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 36 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR 39 This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV. 40 43 44 44 45</type>		
			$46 \\ 47$

	346		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.7.2 No	nblocking Neighborhood Allto	ball
4 5	MPI_INEIG	HBOR_ALLTOALL(sendbuf, second, request)	endcount, sendtype, recvbuf, recvcount, recvtype,
6 7	IN	sendbuf	starting address of send buffer (choice)
8 9	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
10	IN	sendtype	data type of send buffer elements (handle)
11 12	OUT	recvbuf	starting address of receive buffer (choice)
13 14	IN	recvcount	number of elements received from each neighbor (non-negative integer)
15	IN	recvtype	data type of receive buffer elements (handle)
16 17	IN	comm	communicator with topology structure (handle)
18	OUT	request	communication request (handle)
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>MPI_Datatype sendtype, vold* recvour, int recvourt, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.</type></pre>		
41 42 43 44			

	rdispls, recvtype,	endbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, comm, request)
IN	sendbuf	starting address of send buffer (choice)
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 the displacement (relative to sendbuf) 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 recvbuf) at which to place the incoming data from neighbor i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)
MPI_Inei TYPE TYPE INTE TYPE TYPE TYPE	<pre>const int sdis const int recv MPI_Datatype r ghbor_alltoallv(send recvcounts, rd C(*), DIMENSION(), C(*), DIMENSION(), GER, INTENT(IN), ASY recvcounts(*)</pre>	YNCHRONOUS :: sendcounts(*), sdispls(*), , rdispls(*) ENT(IN) :: sendtype, recvtype IN) :: comm NT(OUT) :: request
<pre>MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,</pre>		

12	MPI_INEIC	GHBOR_ALLTOALLW(sendbuf rdispls, recvtypes, comm,	, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, request)	
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)	
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	
7 8 9 10	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)	
11 12 13 14	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)	
15	OUT	recvbuf	starting address of receive buffer (choice)	
16 17 18	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	
19 20 21 22 23	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)	
24 25 26	IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)	
27	IN	comm	communicator with topology structure (handle)	
28 29	OUT	request	communication request (handle)	
30 31 32 33 34 35	<pre>int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[], 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_Bequest *request)</pre>			
36	MPI_Ineig	ghbor_alltoallw(sendbuf,	sendcounts, sdispls, sendtypes, recvbuf,	
37		· • •	recvtypes, comm, request, ierror)	
38 39	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf			
40	<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)</pre>			
41), INTENT(IN), ASYNCHRONOUS ::	
42		<pre>sdispls(*), rdispls</pre>		
43	TYPE(), ASYNCHRONOUS :: sendtypes(*),	
44		recvtypes(*)		
45 46		(MPI_Comm), INTENT(IN) ::		
47		MPI_Request), INTENT(OUT)		
48	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			

<type></type>	RECVCOUNTS, RDISPLS, > SENDBUF(*), RECVBUF(*)	ENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVTYPES, COMM, REQUEST, IERROR)	1 2 3 4
	ER(KIND=MPI_ADDRESS_KIND) ER SENDCOUNTS(*), SENDTYP REQUEST, IERROR	SDISPLS(*), RDISPLS(*) ES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,	5 6
This ca	all starts a nonblocking varian	t of MPI_NEIGHBOR_ALLTOALLW.	7 8 9
7.8 Pers	sistent Neighborhood Co	mmunication on Process Topologies	10 11
mance bene	efits for programs with repeti	collective operations can offer significant perfor- tive communication patterns. The semantics are as described in Section 5.13 .	12 13 14 15
7.8.1 Pers	sistent Neighborhood Gather		16 17 18
MPI_NEIGH	IBOR_ALLGATHER_INIT(sen recvtype, comm, info, req	dbuf, sendcount, sendtype, recvbuf, recvcount, uest)	19 20 21
IN	sendbuf	starting address of send buffer (choice)	22
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	23 24
IN	sendtype	data type of send buffer elements (handle)	25 26
OUT	recvbuf	starting address of receive buffer (choice)	
IN	recvcount	number of elements received from each neighbor (non-negative integer)	27 28 29
IN	recvtype	data type of receive buffer elements (handle)	30
IN	comm	communicator with topology structure (handle)	31 32
IN	info	info argument (handle)	33
OUT	request	communication request (handle)	34
001	request	communication request (narrate)	35
int MPI_Ne	eighbor_allgather_init(co	nst void* sendbuf, int sendcount,	36 37
	MPI_Datatype sendtype	e, void* recvbuf, int recvcount,	38
		e, MPI_Comm comm, MPI_Info info,	39
	MPI_Request *request)		40
MPI_Neight	-	f, sendcount, sendtype, recvbuf,	41
	• •	comm, info, request, ierror)	42
	*), DIMENSION(), INTENT *), DIMENSION(), ASYNCH	(IN), ASYNCHRONOUS :: sendbuf	43 44
	ER, INTENT(IN) :: sendco		45
	<pre>MPI_Datatype), INTENT(IN)</pre>		46
	<pre>/PI_Comm), INTENT(IN) ::</pre>		47
TYPE(M	<pre>MPI_Info), INTENT(IN) ::</pre>	info	48

1 2		(MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	-
3 4 5	MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>		
6 7 8	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR		
9 10 11	Creates a persistent collective communication request for the neighborhood allgather operation.		
12 13 14	MPI_NEIG	HBOR_ALLGATHERV_INIT(s displs, recvtype, comm, ii	endbuf, sendcount, sendtype, recvbuf, recvcounts, nfo, request)
15	IN	sendbuf	starting address of send buffer (choice)
16 17 18	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
19	IN	sendtype	data type of send buffer elements (handle)
20	OUT	recvbuf	starting address of receive buffer (choice)
21 22 23 24	IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor
25 26 27	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
28	IN	recvtype	data type of receive buffer elements (handle)
29 30	IN	comm	communicator with topology structure (handle)
31	IN	info	info argument (handle)
32 33	OUT	request	communication request (handle)
34 35 36 37 38	<pre>int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>		
39 40	MPI_Neigh	0	buf, sendcount, sendtype, recvbuf, recvtype, comm, info, request, ierror)
41	TYPE	(*), DIMENSION(), INTEN	
42		(*), DIMENSION(), ASYNC	
43		ER, INTENT(IN) :: sendco	ount NOUS :: recvcounts(*), displs(*)
44 45		(MPI_Datatype), INTENT(IN)	-
46		(MPI_Comm), INTENT(IN) ::	
47		(MPI_Info), INTENT(IN) ::	
48	TYPE (MPI_Request), INTENT(OUT)) :: request

7.8.	PERSISTENT NEIGHBORHOOI	O COMMUNICATION	351
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
мрт	NETCHBOR ALLCATHERV INIT(SEND	BUF, SENDCOUNT, SENDTYPE, RECVBUF,	2
· · · · · _		RECVTYPE, COMM, INFO, REQUEST, IERROR)	3
	<type> SENDBUF(*), RECVBUF(*)</type>		4
	INTEGER SENDCOUNT, SENDTYPE,	RECVCOUNTS(*), DISPLS(*), RECVTYPE, COM	
	INFO, REQUEST, IERR	OR	7
	Creates a persistent collective com	nunication request for the neighborhood allgat	herv ⁸
	ation.		9
_			10
7.8.2	Persistent Neighborhood Alltoa	II	11
			12 13
			13
MPI.	NEIGHBOR_ALLTOALL_INIT(send recvtype, comm, info, rec	buf, sendcount, sendtype, recvbuf, recvcount, quest)	15 16
IN	sendbuf	starting address of send buffer (choice)	10
IN	sendcount	number of elements sent to each neighbor (non-neg	vative ¹⁸
	Schucount	integer)	19
IN	sendtype	data type of send buffer elements (handle)	20
OL		starting address of receive buffer (choice)	21
			22 23
IN	recvcount	number of elements received from each neighbor (non- 23 24
		negative integer)	25
IN	recvtype	data type of receive buffer elements (handle)	26
IN	comm	communicator with topology structure (handle)	27
IN	info	info argument (handle)	28
οι	T request	communication request (handle)	29
			30 31
int	MPI_Neighbor_alltoall_init(co	nst void* sendbuf, int sendcount,	32
	MPI_Datatype sendtyp	e, void* recvbuf, int recvcount,	33
	· · · · · ·	e, MPI_Comm comm, MPI_Info info,	34
	MPI_Request *request	.)	35
MPI_	Neighbor_alltoall_init(sendbu	f, sendcount, sendtype, recvbuf,	36
	recvcount, recvtype,	comm, info, request, ierror)	37
	<pre>TYPE(*), DIMENSION(), INTEN</pre>		38
	<pre>TYPE(*), DIMENSION(), ASYNC</pre>		39 40
	INTEGER, INTENT(IN) :: sendc		40
	TYPE(MPI_Datatype), INTENT(IN		42
	<pre>TYPE(MPI_Comm), INTENT(IN) :: TYPE(MPI_Info), INTENT(IN) ::</pre>		43
	TYPE(MPI_THIO), INTENT(IN) TYPE(MPI_Request), INTENT(OUT		44
	INTEGER, OPTIONAL, INTENT(OUT	-	45
мът			46
mP1_		F, SENDCOUNT, SENDTYPE, RECVBUF, COMM, INFO, REQUEST, IERROR)	47
		Joining 1111 0, 100 000 1, 101000000	48

1 2 3 4	<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR</type>		
5 6	Creates a persistent collective communication request for the neighborhood all toall operation.		
7 8 9	MPI_NEIGHBOR_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request)		
10 11	IN	sendbuf	starting address of send buffer (choice)
12 13	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
14 15 16	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
17 18	IN	sendtype	data type of send buffer elements (handle)
19	OUT	recvbuf	starting address of receive buffer (choice)
20 21 22	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
23 24 25 26	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
27	IN	recvtype	data type of receive buffer elements (handle)
28	IN	comm	communicator with topology structure (handle)
29 30	IN	info	info argument (handle)
31	OUT	request	communication request (handle)
32 33 34 35 36 37	int MPI_	const int send MPI_Datatype s const int rdis	_init(const void* sendbuf, dcounts[], const int sdispls[], sendtype, void* recvbuf, const int recvcounts[], spls[], MPI_Datatype recvtype, MPI_Comm comm, , MPI_Request *request)
38 39 40	MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request,		
41 42 43 44 45 46	<pre>ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*) TYPE(MPI Datatype) INTENT(IN) :: sendtype recytype</pre>		
46 47 48	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info		

TYPE(MPI_Request), INTENT(OUT) :: request1INTEGER, OPTIONAL, INTENT(OUT) :: ierror2				
<pre>MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, INFO, REQUEST, IERROR</type></pre>				
Creat operation	_	nmunication request for the neighborhood all toallv	10 11 12	
MPI_NEI		endbuf, sendcounts, sdispls, sendtypes, recvbuf, types, comm, info, request)	13 14 15	
IN	sendbuf	starting address of send buffer (choice)	16	
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	17 18 19	
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)	20 21 22 23	
IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)	24 25 26	
OUT	recvbuf	starting address of receive buffer (choice)	27 28	
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	29 30 31	
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)	32 33 34 35	
IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)	36 37 38 39	
IN	comm	communicator with topology structure (handle)	40	
IN	info	info argument (handle)	41	
OUT	request	communication request (handle)	42 43	
<pre>44 int MPI_Neighbor_alltoallw_init(const void* sendbuf,</pre>				

1	<pre>const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,</pre>
2	MPI_Request *request)
3	MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
4	recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
5	ierror)
6	
7	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf
8	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf
9	INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
10	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
11	<pre>sdispls(*), rdispls(*)</pre>
12	TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
13	recvtypes(*)
14	TYPE(MPI_Comm), INTENT(IN) :: comm
15	TYPE(MPI_Info), INTENT(IN) :: info
	TYPE(MPI_Request), INTENT(OUT) :: request
16	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17	,,,,,
18	MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
19	RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
20	IERROR)
21	<type> SENDBUF(*), RECVBUF(*)</type>
22	INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
23	INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
24	INFO, REQUEST, IERROR
25	
26	Creates a persistent collective communication request for the neighborhood all toallw

Creates a persistent collective communication request for the neighborhood alltoallw operation.

7.9 An Application Example

Example 7.9 The example in Figures 7.2-7.5 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)).

43

27 28 29

- 44
- 45
- 46 47
- 48

```
6
                                                                                    7
INTEGER ndims, num_neigh
                                                                                    8
LOGICAL reorder
                                                                                    9
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                    10
INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
                                                                                    11
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
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
                                                                                    17
    Set process grid size and periodicity
Т
                                                                                    18
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
                                                                                    19
periods(1) = .TRUE.
                                                                                    20
periods(2) = .TRUE.
                                                                                    21
    Create a grid structure in WORLD group and inquire about own position
!
                                                                                    22
CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
                                                                                    23
                       comm_cart, ierr)
                                                                                    24
CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords, ierr)
                                                                                    25
i = own_coords(1)
                                                                                    26
j = own_coords(2)
                                                                                    27
! Look up the ranks for the neighbors. Own process coordinates are (i,j).
                                                                                    28
! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
                                                                                    29
CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
                                                                                    30
CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
                                                                                    31
! Initialize the grid functions and start the iteration
                                                                                    32
CALL init (u, f)
                                                                                    33
DO it=1,100
                                                                                    34
   CALL relax (u, f)
                                                                                    35
!
       Exchange data with neighbor processes
                                                                                    36
   CALL exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                    37
END DO
                                                                                    38
CALL output (u)
                                                                                    39
                                                                                    40
                                                                                    41
   Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
```

```
1
\mathbf{2}
3
4
5
6
7
8
9
10
     SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
11
     REAL u(0:101,0:101)
12
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
13
     REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
14
     INTEGER ierr
15
     sndbuf(1:100,1) = u( 1,1:100)
16
     sndbuf(1:100,2) = u(100,1:100)
17
     sndbuf(1:100,3) = u(1:100, 1)
18
     sndbuf(1:100,4) = u(1:100,100)
19
     CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
20
                                    comm_cart, ierr)
21
     ! instead of
22
     ! DO i=1,num_neigh
23
         CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),&
     !
24
     !
                          ierr)
25
         CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),&
     !
26
     !
                          ierr)
27
     ! END DO
28
     ! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
29
30
     u( 0,1:100) = rcvbuf(1:100,1)
^{31}
     u(101,1:100) = rcvbuf(1:100,2)
32
     u(1:100, 0) = rcvbuf(1:100,3)
33
     u(1:100,101) = rcvbuf(1:100,4)
34
     END
35
36
37
     Figure 7.3: Communication routine with local data copying and sparse neighborhood all-
38
     to-all.
39
40
41
42
43
44
45
46
47
48
```

```
3
                                                                                    4
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
                                                                                    5
IMPLICIT NONE
                                                                                    6
USE MPI
                                                                                    7
REAL u(0:101,0:101)
                                                                                     8
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                    9
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
                                                                                    10
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
                                                                                    11
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
                                                                                    12
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
                                                                                    13
INTEGER type_vec, ierr
                                                                                    14
! The following initialization need to be done only once
                                                                                    15
! before the first call of exchange.
                                                                                    16
CALL MPI_TYPE_GET_EXTENT (MPI_REAL, lb, sizeofreal, ierr)
                                                                                    17
CALL MPI_TYPE_VECTOR (100, 1, 102, MPI_REAL, type_vec, ierr)
                                                                                    18
CALL MPI_TYPE_COMMIT (type_vec, ierr)
                                                                                    19
sndtypes(1:2) = type_vec
                                                                                    20
sndcounts(1:2) = 1
                                                                                    21
sndtypes(3:4) = MPI_REAL
                                                                                    22
sndcounts(3:4) = 100
                                                                                    23
rcvtypes = sndtypes
                                                                                    24
rcvcounts = sndcounts
                                                                                    25
sdispls(1) = ( 1 + 1*102) * sizeofreal ! first element of u( 1
                                                                           , 1:100)<sub>26</sub>
                                                                            1:100) 27
sdispls(2) = (100 + 1*102) * size of real ! first element of u(100)
sdispls(3) = ( 1 + 1*102) * sizeofreal ! first element of u( 1:100,
                                                                              1
                                                                                   )<sub>28</sub>
sdispls(4) = (1 + 100*102) * sizeofreal ! first element of u( 1:100,100
                                                                                   )<sub>29</sub>
rdispls(1) = ( 0 + 1*102) * sizeofreal ! first element of u( 0
                                                                           , 1:100)<sub>30</sub>
                                                                             1:100)<sub>31</sub>
rdispls(2) = (101 + 1*102) * size of real ! first element of u(101)
rdispls(3) = ( 1 + 0*102) * sizeofreal ! first element of u( 1:100,
                                                                              0
                                                                                   )<sub>32</sub>
rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u( 1:100,101
                                                                                   )33
! the following communication has to be done in each call of exchange
                                                                                    34
CALL MPI_NEIGHBOR_ALLTOALLW (u, sndcounts, sdispls, sndtypes, &
                                                                                    35
                               u, rcvcounts, rdispls, rcvtypes, &
                                                                                    36
                               comm_cart, ierr)
                                                                                    37
! The following finalizing need to be done only once
                                                                                    38
! after the last call of exchange.
                                                                                    39
CALL MPI_TYPE_FREE (type_vec, ierr)
                                                                                    40
END
                                                                                    41
                                                                                    42
```

Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local data copying.

```
    43
    44
    45
    46
    47
    48
```

```
INTEGER ndims, num_neigh
1
     LOGICAL reorder
2
    PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
3
     INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
4
    LOGICAL periods(ndims)
5
    REAL u(0:101,0:101), f(0:101,0:101)
6
    DATA dims / ndims * 0 /
7
     INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
8
     INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
9
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
10
     INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
11
     INTEGER type_vec, request, status
12
     comm = MPI_COMM_WORLD
13
     CALL MPI_COMM_SIZE(comm, comm_size, ierr)
14
         Set process grid size and periodicity
15
    CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
16
    periods(1) = .TRUE.
17
    periods(2) = .TRUE.
18
         Create a grid structure in WORLD group
     !
19
     CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
20
                            comm_cart, ierr)
21
     ! Create datatypes for the neighborhood communication
22
     !
23
     ! Insert code from example in Figure 7.4 to create and initialize
24
     ! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
25
     Ţ
26
     ! Initialize the neighborhood all-to-all-w operation
27
     CALL MPI_NEIGHBOR_ALLTOALLW_INIT (u, sndcounts, sdispls, sndtypes, &
28
                                         u, rcvcounts, rdispls, rcvtypes, &
29
                                         comm_cart, info, request, ierr)
30
     ! Initialize the grid functions and start the iteration
31
     CALL init (u, f)
32
     DO it=1,100
33
            Start data exchange with neighbor processes
     !
34
        CALL MPI_START( request, ierr)
35
            Compute inner cells
     !
36
        CALL relax_inner (u, f)
37
     !
            Check on completion of neighbor exchange
38
        CALL MPI_WAIT( request, status, ierr)
39
            Compute edge cells
     1
40
        CALL relax_edges (u, f)
41
     END DO
42
     CALL output (u)
43
     CALL MPI_REQUEST_FREE (request, ierr)
44
     CALL MPI_TYPE_FREE (type_vec, ierr)
45
46
47
     Figure 7.5: Two-dimensional parallel Poisson solver with persistent sparse neighborhood
```

⁴⁸ all-to-all-w and without local data copying.

Chapter 8

MPI Environmental Management

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

> 6 7

> 9

15

16

17

1819

2021

22

23

 24 25

26

2728

29

30 31

Implementation Information 8.1

8.1.1 Version Inquiries

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

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

#define MPI_VERSION 3 #define MPI_SUBVERSION 1

in	Fortran,			
111	ronnan,		32	
INTEGER :: MPI_VERSION, MPI_SUBVERSION				
	PARAMETER (MPI_	VERSION = 3)	34	
	PARAMETER (MPI_	SUBVERSION = 1)	35	
			36	
	For runtime determ	nination,	37	
			38	
М	PI_GET_VERSION(ve	ersion subversion)	39	
			40	
(OUT version	version number (integer)	41	
(OUT subversion	subversion number (integer)	42	
			43	
in	t MPT Get version	(int *version, int *subversion)	44	
MP	MPI_Get_version(version, subversion, ierror) 4			
	INTEGER, INTENT	(OUT) :: version, subversion	47	
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48			

	900		CHAFTER 8. WFTENVIRONMENTAL MANAGEMENT
1 2		VERSION(VERSION, EGER VERSION, SUBV	SUBVERSION, IERROR) VERSION, IERROR
3 4 5 6 7 8	function MPI_SUB	must always be thr	be called before MPI_INIT and after MPI_FINALIZE. This read-safe, as defined in Section 12.4. Valid (MPI_VERSION, is and previous versions of the MPI standard are (3,1), (3,0),
9	MPI_GET	_LIBRARY_VERSIO	N(version, resultlen)
10 11	OUT	version	version string (string)
12 13 14	OUT	resultlen	Length (in printable characters) of the result returned in version (integer)
5	int MPI_	_Get_library_versi	ion(char *version, int *resultlen)
6 7 8 9 0	CHAR INTE	v	
1 2 3	CHAF	LIBRARY_VERSION(V ACTER*(*) VERSION EGER RESULTLEN,IEF	
4 5 6			ring representing the version of the MPI library. The version g for maximum flexibility.
7 3 9 0	for	-	. An implementation of MPI should return a different string source code or build that could be visible to the user. (<i>End of</i>)
	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. 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_STRING. MPI_GET_LIBRARY_VERSION can be called before MPI_INIT and after MPI_FINALIZE. This function must always be thread-safe, as defined in Section 12.4.		
2	8.1.2 E	nvironmental Inquiri	es
	cator MP inquired	I_COMM_WORLD wh by using the function	ibe the execution environment are attached to the communi- tion MPI is initialized. The values of these attributes can be on MPI_COMM_GET_ATTR described in Section 6.7 and in its to delete these attributes, free their keys, or change their

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

The list of predefined attribute keys include	1			
MPI_TAG_UB Upper bound for tag value.	2 3			
MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.	4			
MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.	5 6 7			
MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.	8			
Vendors may add implementation-specific parameters (such as node number, real mem- ory size, virtual memory size, etc.) These predefined attributes do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	9 10 11 12 13			
Advice to users. Note that in the C binding, the value returned by these attributes is a <i>pointer</i> to an int containing the requested value. (<i>End of advice to users.</i>)	14 15			
The required parameter values are discussed in more detail below:	16 17			
Tag Values	18			
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.				
Host Rank	27			
The value returned for MPI_HOST gets the rank of the <i>HOST</i> process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a <i>HOST</i> , nor does it requires that a <i>HOST</i> exists. The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	28 29 30 31 32 33			
IO Rank	34 35			
The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ISO C I/O operations are supported (e.g., fopen, fprintf, lseek). If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O, then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value MPI_PROC_NULL will be returned.	36 37 38 39 40 41 42 43 44 45 46			
Advice to users. Note that input is not collective, and this attribute does not indicate	47			

Advice to users. Note that input is not collective, and this attribute does not indicate which process can or does provide input. (End of advice to users.)

Unofficial Draft for Comment Only

1	Clock Synchronization			
2 3 4 5 6 7 8 9 10 11 12 13 14	The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less then one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one. The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD. The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of MPI_COMM_WORLD.			
15 16	Inquire Processor Name			
17 18				
19	MPI_GET_PROCESSOR_NAME(name, resultlen)			
20 21	OUT	name	A unique specifier for the actual (as opposed to virtual) node.	
22 23 24	OUT	resultlen	Length (in printable characters) of the result returned in name	
25 26	<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>			
27 28 29 30 31	<pre>MPI_Get_processor_name(name, resultlen, ierror) CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR) CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR</pre>			
32 33 34				
35 36 37 38 39 40 41 42 43 44 45 46	ne of the processor on which it was called at the moment acter string for maximum flexibility. From this value it ecific piece of hardware; possible values include "processor "231" (where 231 is the actual processor number in the The argument name must represent storage that is at least aracters long. MPI_GET_PROCESSOR_NAME may write name. cually written is returned in the output argument, resultlen. ally stored at name[resultlen]. The value of resultlen cannot SOR_NAME-1. In Fortran, name is padded on the right with ultlen cannot be larger than MPI_MAX_PROCESSOR_NAME.			
47 48			allows MPI implementations that do process migration to or. Note that nothing in MPI <i>requires</i> or defines process	

migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

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

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 11.5.3.

MPI_ALLOC_MEM(size, info, baseptr)

	(
IN	size	size of memory segment in bytes (non-negative inte-	21
		ger)	22
IN	info	info argument (handle)	23
IIN	iiio	into argument (nandie)	24
OUT	baseptr	pointer to beginning of memory segment allocated	25
			26
int MPI_A	lloc_mem(MPI_Aint size, N	<pre>IPI_Info info, void *baseptr)</pre>	27
		- -	28
	_mem(size, info, baseptr,		29
USE,	INTRINSIC :: ISO_C_BINDI	ING, ONLY : C_PTR	30
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size TYPE(MPI_Info), INTENT(IN) :: info			31
			32
TYPE(C_PTR), INTENT(OUT) :: h	baseptr	33
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	34
MPT ALLOC	MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)		
	ER INFO, IERROR		36
	ER(KIND=MPI_ADDRESS_KIND)	ST7F BASEPTR	37
INIDO			38
If the	Fortran compiler provides TYP	E(C_PTR), then the following generic interface must	39
be provide	ed in the mpi module and she	ould be provided in mpif.h through overloading,	40
i.e., with t	he same routine name as the	routine with INTEGER(KIND=MPI_ADDRESS_KIND)	41
BASEPTR, but with a different specific procedure name:			

INTERFACE MPI_ALLOC_MEM	44
SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)	4
IMPORT :: MPI_ADDRESS_KIND	40
INTEGER INFO, IERROR	4'
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	48

Unofficial Draft for Comment Only

```
1
          END SUBROUTINE
2
          SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
3
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
4
              IMPORT :: MPI_ADDRESS_KIND
5
              INTEGER :: INFO, IERROR
6
              INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
7
              TYPE(C_PTR) :: BASEPTR
8
          END SUBROUTINE
9
     END INTERFACE
10
          The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
11
     implied specific procedure names are described in Section 18.1.5.
12
          The info argument can be used to provide directives that control the desired location
13
     of the allocated memory. Such a directive does not affect the semantics of the call. Valid
14
     info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL
15
     is always valid.
16
          The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
17
     to indicate it failed because memory is exhausted.
18
19
20
     MPI_FREE_MEM(base)
21
22
       IN
                 base
                                             initial address of memory segment allocated by
23
                                             MPI_ALLOC_MEM (choice)
^{24}
25
     int MPI_Free_mem(void *base)
26
     MPI_Free_mem(base, ierror)
27
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_FREE_MEM(BASE, IERROR)
31
          <type> BASE(*)
32
          INTEGER IERROR
33
          The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to
34
     indicate an invalid base argument.
35
36
           Rationale. The C bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar
37
           to the bindings for the malloc and free C library calls: a call to
38
           MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one
39
           less level of indirection). Both arguments are declared to be of same type
40
           void<sup>*</sup> so as to facilitate type casting. The Fortran binding is consistent with the C
41
           bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the TYPE(C_PTR)
42
           pointer or the (integer valued) address of the allocated memory. The base argument
43
           of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable
44
           stored at that location. (End of rationale.)
45
46
           Advice to implementors.
                                      If MPI_ALLOC_MEM allocates special memory, then a
47
           design similar to the design of C malloc and free functions has to be used, in order
48
```

Example 8.1

TYPE(C_PTR) poi

to find out the size of a memory segment, when the segment is freed. If no special 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. (<i>End of advice to implementors.</i>)
Example 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with PE(C_PTR) pointers. We assume 4-byte REALs.
JSE mpi_f08 ! or USE mpi (not guaranteed with INCLUDE 'mpif.h') JSE, INTRINSIC :: ISO_C_BINDING FYPE(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
! in ISO_C_BINDING
a(3,5) = 2.71;
CALL MPI_Free_mem(a, ierr) ! memory is freed

Example 8.2 Example of use of MPI_ALLOC_MEM, in Fortran with non-standard Craypointers. We assume 4-byte REALs, and assume that these pointers are address-sized.

REAL A POINTER (P, A(100,100)) ! no memory is allocated INTEGER(KIND=MPI_ADDRESS_KIND) SIZE SIZE = 4*100*100CALL 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

 $\mathbf{2}$

3

4

5

6 7

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

 24 25

26

2728

29

30

31

32

33

34

35

36

37

38

39

40 41

42

43

44

45

48

```
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 implementation-dependent. Each such error generates an **MPI exception**.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled. More background information about how MPI treats errors can be found in Section 2.8.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

- MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort all connected processes. This is similar to calling MPI_ABORT using a communicator containing all connected processes with an implementation-specific value as the errorcode argument.
- MPI_ERRORS_ABORT The handler, when called, is invoked on a communicator in a manner similar to calling MPI_ABORT on that communicator. If the error handler is invoked on an window or a file, it is similar to calling MPI_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. In either case, the value that would be provided as the errorcode argument to MPI_ABORT is implementation-specific.
 - **MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

Advice to implementors. The implementation-specific error information resulting from MPI_ERRORS_ARE_FATAL and MPI_ERRORS_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (End of advice to implementors.)

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

11 12

13

14

15

26

38

39

40 41

42

43

44

After initialization, the error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if any). Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non-trivial MPI error handler.

After an error is detected, MPI will provide the user as much information as possible about that error using error classes. Some errors might prevent MPI from completing further API calls successfully and those functions will continue to report errors until the cause of the error is corrected or the user terminates the application. The user can make the determination of whether or not to attempt to continue after detecting such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

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 and available recovery actions. (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, where XXX is, respectively, COMM, WIN, or FILE.

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 implementations should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10 11

12

13

14

15

16

17

18 19

20

21

22

23

24 25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

 $46 \\ 47$

1 2 3 4	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. (<i>End of advice to implementors.</i>)			
5	The syntax for these calls is given below.			
6 7 8 9	8.3.1 Error Handlers for Communicators			
10	MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)			
11 12	IN comm_errhandler_fn user defined error handling procedure (function)			
13 14	OUTerrhandlerMPI error handler (handle)			
15 16	<pre>int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *comm_errhandler_fn, MPI_Errhandler *errhandler)</pre>			
17 18 19 20 21	<pre>MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
22 23 24 25	MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL COMM_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR			
26 27 28	Creates an error handler that can be attached to communicators. The user routine should be, in C, a function of type MPI_Comm_errhandler_function, which is defined as			
29	<pre>typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *,);</pre>			
30 31 32 33 34 35 36 37	The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be of the form:			
38 39 40 41	ABSTRACT INTERFACE SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) TYPE(MPI_Comm) :: comm INTEGER :: error_code			
42 43 44 45 46 47 48	With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN should be of the form: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE			

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

Rationale. The variable argument list is provided because it provides an ISOstandard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

Advice to users. A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify a "global" error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to* users.)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)			12	
	× ×		13	
INOUT	comm	communicator (handle)	14	
IN	errhandler	new error handler for communicator (handle)	15	
			16	
int MPI_C	Comm_set_errhandler(MPI_Co	omm comm, MPI_Errhandler errhandler)	17	
MPT Comm	_set_errhandler(comm, err]	handler jerror)	18 19	
	(MPI_Comm), INTENT(IN) ::	-	20	
	(MPI_Errhandler), INTENT()		21	
	ER, OPTIONAL, INTENT(OUT)		22	
NDT 00104			23	
	SET_ERRHANDLER(COMM, ERRI		24	
INTEG	ER COMM, ERRHANDLER, IER	KUK	25	
Attac	hes a new error handler to a	communicator. The error handler must be either	26	
*	,	r handler created by a call to	27	
MPI_COM	M_CREATE_ERRHANDLER.		28	
			29	
	M_GET_ERRHANDLER(comn	a errhandler)	30	
	```		31	
IN	comm	communicator (handle)	32	
OUT	errhandler	error handler currently associated with communicator	33 34	
		(handle)	35	
			36	
int MPI_C	Comm_get_errhandler(MPI_C	omm comm, MPI_Errhandler *errhandler)	37	
MPT Comm	_get_errhandler(comm, err]	handler. jerror)	38	
	(MPI_Comm), INTENT(IN) ::		39	
	(MPI_Errhandler), INTENT(		40	
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	41	
MDT COMM			42	
	_GET_ERRHANDLER(COMM, ERRI GER COMM, ERRHANDLER, IERI	-	43	
	ER COMP, ERRIANDEER, TER		44	
		y associated with a communicator.	45	
	2, 0	register at its entry point the current error handler	46	
	,	e error handler for this communicator, and restore	47	
before exit	before exiting the previous error handler. 4			

#### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

```
370
 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT
1
 8.3.2
 Error Handlers for Windows
\mathbf{2}
3
4
 MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)
5
 IN
 win_errhandler_fn
 user defined error handling procedure (function)
6
 OUT
7
 errhandler
 MPI error handler (handle)
8
9
 int MPI_Win_create_errhandler(MPI_Win_errhandler_function
10
 *win_errhandler_fn, MPI_Errhandler *errhandler)
11
 MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
12
 PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
13
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
 MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
17
 EXTERNAL WIN_ERRHANDLER_FN
18
 INTEGER ERRHANDLER, IERROR
19
 Creates an error handler that can be attached to a window object. The user routine
20
 should be, in C, a function of type MPI_Win_errhandler_function which is defined as
21
 typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
22
23
 The first argument is the window in use, the second is the error code to be returned.
^{24}
 With the Fortran mpi_f08 module, the user routine win_errhandler_fn should be of the form:
25
 ABSTRACT INTERFACE
26
 SUBROUTINE MPI_Win_errhandler_function(win, error_code)
27
 TYPE(MPI_Win) :: win
28
 INTEGER :: error_code
29
 With the Fortran mpi module and mpif.h, the user routine WIN_ERRHANDLER_FN should
30
 be of the form:
^{31}
 SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
32
 INTEGER WIN, ERROR_CODE
33
34
35
36
 MPI_WIN_SET_ERRHANDLER(win, errhandler)
37
 INOUT
 win
 window (handle)
38
 IN
 errhandler
 new error handler for window (handle)
39
40
 int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
41
42
 MPI_Win_set_errhandler(win, errhandler, ierror)
43
 TYPE(MPI_Win), INTENT(IN) :: win
44
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
 MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
48
 INTEGER WIN, ERRHANDLER, IERROR
```

**Unofficial Draft for Comment Only** 

1 Attaches a new error handler to a window. The error handler must be either a pre- $\mathbf{2}$ defined error handler, or an error handler created by a call to 3 MPI_WIN_CREATE_ERRHANDLER. 4 5 MPI_WIN_GET_ERRHANDLER(win, errhandler) 6 IN 7 win window (handle) 8 OUT errhandler error handler currently associated with window (han-9 dle) 10 11 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler) 1213 MPI_Win_get_errhandler(win, errhandler, ierror) 14TYPE(MPI_Win), INTENT(IN) :: win 15TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR) 18 INTEGER WIN, ERRHANDLER, IERROR 19 20Retrieves the error handler currently associated with a window. 2122 8.3.3 Error Handlers for Files 23 24 25MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler) 26IN file_errhandler_fn user defined error handling procedure (function) 2728 OUT errhandler MPI error handler (handle) 2930 int MPI_File_create_errhandler(MPI_File_errhandler_function 31*file_errhandler_fn, MPI_Errhandler *errhandler) 32 33 MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn 34 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR) 38 EXTERNAL FILE_ERRHANDLER_FN 39 INTEGER ERRHANDLER, IERROR 40 41 Creates an error handler that can be attached to a file object. The user routine should 42be, in C, a function of type MPI_File_errhandler_function, which is defined as typedef void MPI_File_errhandler_function(MPI_File *, int *, ...); 43 44The first argument is the file in use, the second is the error code to be returned. 45With the Fortran mpi_f08 module, the user routine file_errhandler_fn should be of the form: 46 ABSTRACT INTERFACE 47

SUBROUTINE MPI_File_errhandler_function(file, error_code)

Unofficial Draft for Comment Only

```
1
 TYPE(MPI_File) :: file
\mathbf{2}
 INTEGER :: error_code
3
 With the Fortran mpi module and mpif.h, the user routine FILE_ERRHANDLER_FN should
4
 be of the form:
5
 SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
6
 INTEGER FILE, ERROR_CODE
7
8
9
 MPI_FILE_SET_ERRHANDLER(file, errhandler)
10
11
 INOUT
 file
 file (handle)
12
 IN
 errhandler
 new error handler for file (handle)
13
14
 int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
15
16
 MPI_File_set_errhandler(file, errhandler, ierror)
17
 TYPE(MPI_File), INTENT(IN) :: file
18
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
21
 INTEGER FILE, ERRHANDLER, IERROR
22
23
 Attaches a new error handler to a file. The error handler must be either a predefined
^{24}
 error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
25
26
 MPI_FILE_GET_ERRHANDLER(file, errhandler)
27
28
 IN
 file
 file (handle)
29
 OUT
 errhandler
 error handler currently associated with file (handle)
30
^{31}
32
 int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
33
 MPI_File_get_errhandler(file, errhandler, ierror)
34
 TYPE(MPI_File), INTENT(IN) :: file
35
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
38
 INTEGER FILE, ERRHANDLER, IERROR
39
40
 Retrieves the error handler currently associated with a file.
41
42
43
44
45
46
47
48
```

8.3.	ERR	OR HANDLING		373
8.3.	4 Fre	eing Errorhandlers and Retrie	eving Error Strings	1
				3
MPI	_ERRH	ANDLER_FREE(errhandler)		4
	Ουτ	errhandler	MPI error handler (handle)	5
	001	cimandici		6
int	мрт ғ	Crrhandler_free(MPI_Errha	ndler *errhandler)	7
				9
MPI.		ndler_free(errhandler, i		10
		MPI_Errhandler), INTENT( ER, OPTIONAL, INTENT(OUT		11
	TNIEG	ER, OFIIONAL, INIENI(001	) 101101	12
MPI.		NDLER_FREE(ERRHANDLER, I	ERROR)	13
	INTEG	ER ERRHANDLER, IERROR		14
	Marks	the error handler associated	with errhandler for deallocation and sets errha	ndler 15
			handler will be deallocated after all the ob-	$_{17}$
asso	ciated	with it (communicator, wind	ow, or file) have been deallocated.	18
				19
MPI	_ERRC	OR_STRING(errorcode, string,	resultlen)	20
IN		errorcode	Error code returned by an MPI routine	21
				22
	UT	string	Text that corresponds to the errorcode	23
0	UT	resultlen	Length (in printable characters) of the result ret	urned $^{24}_{25}$
			in string	25 26
•				27
int	MP1_E	<pre>irror_string(int errorcod</pre>	e, char *string, int *resultlen)	28
MPI.	_Error	_string(errorcode, strin	g, resultlen, ierror)	29
		ER, INTENT(IN) :: error		30
		CTER(LEN=MPI_MAX_ERROR_S	0	31
		ER, INTENT(OUT) :: resu		32
	INIEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	33
MPI.	_ERROR	_STRING(ERRORCODE, STRIN	G, RESULTLEN, IERROR)	34 35
		ER ERRORCODE, RESULTLEN,	IERROR	36
	CHARA	CTER*(*) STRING		37
	Return	ns the error string associated	with an error code or class. The argument	string 38
mus	t repre	esent storage that is at least $N$	IPI_MAX_ERROR_STRING characters long.	- 39
	The n	umber of characters actually w	written is returned in the output argument, ${\sf resu}$	Iltlen. 40
	ית			41
			ion was chosen to make the Fortran and C bin-	0
			pointer to a string has two difficulties. First	
	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			-
	-		an, a function declared as returning CHARACTE	
		_ ,	aple, a PRINT statement. (End of rationale.)	47
				48

### Unofficial Draft for Comment Only

1	8.4 E	rror Codes and C	lasses	
2 3 4 5	of MPI_S as possib	UCCESS). This is don ble in the error code	MPI are left entirely to the implementation (with the exception the to allow an implementation to provide as much information (for use with MPI_ERROR_STRING).	
6 7 8 9	MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called <i>error classes</i> . Valid error classes are shown in Table 8.1 and Table 8.2.			
10 11 12 13	string as error coo	sociated with an erro	on MPI_ERROR_STRING can be used to compute the error r class. The values defined for MPI error classes are valid MPI	
14	The		$CESS < MPI_ERR \le MPI_ERR_LASTCODE.$	
15 16 17		tionale. The differen	ace between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that can return useful information about MPI_ERR_OTHER.	
18 19 20 21 22	rat	ion of error classes a	5 = 0 is necessary to be consistent with C practice; the sepa- and error codes allows us to define the error classes this way. ODE is often a nice sanity check as well. ( <i>End of rationale.</i> )	
23 24	MPI_ERI	ROR_CLASS(errorcod	le, errorclass)	
25 26	IN	errorcode	Error code returned by an MPI routine	
27	OUT	errorclass	Error class associated with errorcode	
28 29	int MPI	_Error_class(int (	errorcode, int *errorclass)	
30 31 32 33	MPI_Error_class(errorcode, errorclass, ierror) INTEGER, INTENT(IN) :: errorcode INTEGER, INTENT(OUT) :: errorclass			
34		EGER, OPTIONAL, I		
35 36		UR_CLASS(ERRURCUD) EGER ERRORCODE, EI	E, ERRORCLASS, IERROR) RRORCLASS, IERROR	
37 38 39	The itself.	function MPI_ERRC	$DR_CLASS$ maps each standard error code (error class) onto	
40 41 42	8.5 E	rror Classes, Erro	r Codes, and Error Handlers	
43 44 45	this libra	ary may have its own	yered library on top of an existing MPI implementation, and a set of error codes and classes. An example of such a library PI, see Chapter 13. For this purpose, functions are needed to:	
46 47	1. ado	d a new error class to	the ones an MPI implementation already knows.	
48	2. ass	ociate error codes wi	th this error class, so that $MPI_ERROR_CLASS$ works.	
		Unc	official Draft for Comment Only	

		2
MPI_SUCCESS	No error	3
MPI_ERR_BUFFER	Invalid buffer pointer	4
MPI_ERR_COUNT	Invalid count argument	5
MPI_ERR_TYPE	Invalid datatype argument	6
MPI_ERR_TAG	Invalid tag argument	7
MPI_ERR_COMM	Invalid communicator	8
MPI_ERR_RANK	Invalid rank	9
MPI_ERR_REQUEST	Invalid request (handle)	10
MPI_ERR_ROOT	Invalid root	11
MPI_ERR_GROUP	Invalid group	12
MPI_ERR_OP	Invalid operation	13
MPI_ERR_TOPOLOGY	Invalid topology	14
MPI_ERR_DIMS	Invalid dimension argument	15
MPI_ERR_ARG	Invalid argument of some other kind	16
MPI_ERR_UNKNOWN	Unknown error	17
MPI_ERR_TRUNCATE	Message truncated on receive	18
MPI_ERR_OTHER	Known error not in this list	19
MPI_ERR_INTERN	Internal MPI (implementation) error	20
MPI_ERR_IN_STATUS	Error code is in status	20
MPI_ERR_PENDING	Pending request	22
MPI_ERR_KEYVAL	Invalid keyval has been passed	23
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory	23
	is exhausted	25
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	26
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY	27
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL	28
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	29
MPI_ERR_SPAWN	Error in spawning processes	30
MPI_ERR_PORT	Invalid port name passed to	31
	MPI_COMM_CONNECT	32
MPI_ERR_SERVICE	Invalid service name passed to	33
	MPI_UNPUBLISH_NAME	34
MPI_ERR_NAME	Invalid service name passed to	35
	MPI_LOOKUP_NAME	36
MPI_ERR_WIN	Invalid win argument	37
MPI_ERR_SIZE	Invalid size argument	38
MPI_ERR_DISP	Invalid disp argument	39
MPI_ERR_INFO	Invalid info argument	40
MPI_ERR_LOCKTYPE	Invalid locktype argument	41
MPI_ERR_ASSERT	Invalid assert argument	42
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window	43
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls	43
		45
<b>m 11</b> 0	1. Ennon classes (Dant 1)	46
	IT HEREOR ALAGGOG ( Port 1)	10

Table 8.1: Error classes (Part 1)

**Unofficial Draft for Comment Only** 

1 2 3 4 MPI_ERR_RMA_RANGE Target memory is not part of the win-5dow (in the case of a window created 6 with MPI_WIN_CREATE_DYNAMIC, tar-7 get memory is not attached) 8 Memory cannot be attached (e.g., because MPI_ERR_RMA_ATTACH 9 of resource exhaustion) 10 Memory cannot be shared (e.g., some pro-MPI_ERR_RMA_SHARED 11 cess in the group of the specified commu-12nicator cannot expose shared memory) 13 MPI_ERR_RMA_FLAVOR Passed window has the wrong flavor for the 14 called function 15Invalid file handle MPI_ERR_FILE 16 MPI_ERR_NOT_SAME Collective argument not identical on all 17processes, or collective routines called in 18 a different order by different processes 19 MPI_ERR_AMODE Error related to the **amode** passed to 20MPI_FILE_OPEN 21MPI_ERR_UNSUPPORTED_DATAREP Unsupported datarep passed to 22 MPI_FILE_SET_VIEW 23MPI_ERR_UNSUPPORTED_OPERATION Unsupported operation, such as seeking on 24a file which supports sequential access only 25File does not exist MPI_ERR_NO_SUCH_FILE 26File exists MPI_ERR_FILE_EXISTS 27Invalid file name (e.g., path name too long) MPI_ERR_BAD_FILE 28Permission denied MPI_ERR_ACCESS 29 Not enough space MPI_ERR_NO_SPACE 30 MPI_ERR_QUOTA Quota exceeded 31 Read-only file or file system MPI_ERR_READ_ONLY 32 File operation could not be completed, as MPI_ERR_FILE_IN_USE 33 the file is currently open by some process 34 Conversion functions could not be regis-MPI_ERR_DUP_DATAREP 35 tered because a data representation identi-36 fier that was already defined was passed to 37 MPI_REGISTER_DATAREP 38 MPI_ERR_CONVERSION An error occurred in a user supplied data 39 conversion function. 40 Other I/O error MPI_ERR_IO 41 Last error code MPI_ERR_LASTCODE 4243 44 Table 8.2: Error classes (Part 2) 4546 4748

3. associate strings with these error codes, so that MPI_ERROR_STRING works.

4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes or codes: it is not expected that an application will generate them in significant numbers.

		8
MPI_ADD_ERROR_CLASS(errorclass)		9
OUT errorclass	value for the new error class (integer)	10
		11
<pre>int MPI_Add_error_class(int *error</pre>	class)	12
		13
MPI_Add_error_class(errorclass, ie		14
INTEGER, INTENT(OUT) :: error		15
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	16
MPI_ADD_ERROR_CLASS(ERRORCLASS, IF	RROR)	17
INTEGER ERRORCLASS, IERROR		18
		19
Creates a new error class and return	ns the value for it.	20
Rationale. To avoid conflicts wit	h existing error codes and classes, the value is set	21
by the implementation and not by		22
аў энэ нерокаларыны нерокаларыны аў		23
Advice to implementors. A high	-quality implementation will return the value for	24
a new errorclass in the same deter	ministic way on all processes. (End of advice to	25
implementors.)		26
Advice to users Since a call to MP	I_ADD_ERROR_CLASS is local, the same errorclass	27
	that make this call. Thus, it is not safe to assume	28 29
	t of processes at the same time will yield the same	29 30
0 0	However, if an implementation returns the new	30 31
-	ind they are always generated in the same order on	31 32
chore ass in a deterministic way, a	ia incy are arways generated in the same order on	- 52

However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice* to users.)

the same set of processes (for example, all processes), then the value will be the same.

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new userdefined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change 46 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 48

#### Unofficial Draft for Comment Only

1 2

3 4

 $\mathbf{5}$ 

6

7

33

34

35

36

37

38 39

40

41

42

43

44

```
1
 changed. Note that error codes and error classes are not necessarily dense. A user
\mathbf{2}
 may not assume that each error class below MPI_LASTUSEDCODE is valid. (End of
3
 advice to users.)
4
5
6
 MPI_ADD_ERROR_CODE(errorclass, errorcode)
7
8
 IN
 errorclass
 error class (integer)
9
 OUT
 errorcode
 new error code to associated with errorclass (integer)
10
11
 int MPI_Add_error_code(int errorclass, int *errorcode)
12
13
 MPI_Add_error_code(errorclass, errorcode, ierror)
14
 INTEGER, INTENT(IN) :: errorclass
15
 INTEGER, INTENT(OUT) ::
 errorcode
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
 MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
18
 INTEGER ERRORCLASS, ERRORCODE, IERROR
19
20
 Creates new error code associated with errorclass and returns its value in errorcode.
21
22
 Rationale.
 To avoid conflicts with existing error codes and classes, the value of the
23
 new error code is set by the implementation and not by the user. (End of rationale.)
24
 Advice to implementors.
 A high-quality implementation will return the value for
25
 a new errorcode in the same deterministic way on all processes. (End of advice to
26
 implementors.)
27
28
29
30
 MPI_ADD_ERROR_STRING(errorcode, string)
^{31}
 IN
 errorcode
 error code or class (integer)
32
33
 IN
 string
 text corresponding to errorcode (string)
34
35
 int MPI_Add_error_string(int errorcode, const char *string)
36
 MPI_Add_error_string(errorcode, string, ierror)
37
 INTEGER, INTENT(IN) :: errorcode
38
 CHARACTER(LEN=*), INTENT(IN) :: string
39
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
40
41
 MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
42
 INTEGER ERRORCODE, IERROR
43
 CHARACTER*(*) STRING
44
 Associates an error string with an error code or class. The string must be no more
45
 than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the
46
```

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

47

already has a string will replace the old string with the new string. It is erroneous to call 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).

Section 8.3 describes the methods for creating and associating error handlers with communicators, files, and windows.

MPI_COMM_CALL_ERRHANDLER(comm, errorcode)			
IN	comm	communicator with error handler (handle)	
IN	errorcode	error code (integer)	
<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode) MPI_Comm_call_errhandler(comm, errorcode, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm     INTEGER, INTENT(IN) :: errorcode     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR) INTEGER COMM, ERRORCODE, IERROR			

This function invokes the error handler assigned to the communicator 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).

MPI_WIN_CALL_ERRHANDLER(win, errorcode)

IN	win	window with error handler (handle)	29
			30
IN	errorcode	error code (integer)	31
			32
int	MPI_Win_call_errhandler(MPI_W	n win, int errorcode)	33
мрт	_Win_call_errhandler(win, error	code ierror)	34
···· ±_	TYPE(MPI_Win), INTENT(IN) ::		35
	INTEGER, INTENT(IN) :: error		36
	INTEGER, OPTIONAL, INTENT(OUT)		37
	INIEGEN, OFFICIAL, INTENT(COT)		38
MPI_	_WIN_CALL_ERRHANDLER(WIN, ERROP	RCODE, IERROR)	39
	INTEGER WIN, ERRORCODE, IERROR	2	40

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.)

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

 $45 \\ 46$ 

```
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)
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
41
 MPI_WTIME()
42
43
 double MPI_Wtime(void)
44
45
 DOUBLE PRECISION MPI_Wtime()
46
 DOUBLE PRECISION MPI_WTIME()
47
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 in Section 8.1.2).

```
MPI_WTICK()
```

double MPI_Wtick(void)
DOUBLE PRECISION MPI_Wtick()
DOUBLE PRECISION MPI_WTICK()

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be  $10^{-3}$ .

### 8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

```
MPI_INIT()
int MPI_Init(int *argc, char ***argv)
MPI_Init(ierror)
```

 24 

 $45 \\ 46$ 

```
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
 argy 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, rs6000)

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)
 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 operation.

#### 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.
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. Cancelling a send request by calling MPI_CANCEL is deprecated.

Process 0	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 21to 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, users may desire to supply an exit code for each process that returns from MPI_FINALIZE.

**Example 8.10** The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
. . .
MPI_Finalize();
if (myrank == 0) {
 resultfile = fopen("outfile","w");
 dump_results(resultfile);
 fclose(resultfile);
}
exit(0);
```

1

2

3

4

13

14

15

16

17

18

19

20

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
 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)
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. This function must always
18
 be thread-safe, as defined in Section 12.4.
19
20
21
 MPI_ABORT(comm, errorcode)
22
 IN
 communicator of tasks to abort
 comm
23
 IN
 error code to return to invoking environment
 errorcode
^{24}
25
 int MPI_Abort(MPI_Comm comm, int errorcode)
26
27
 MPI_Abort(comm, errorcode, ierror)
28
 TYPE(MPI_Comm), INTENT(IN) :: comm
29
 INTEGER, INTENT(IN) :: errorcode
30
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
31
 MPI_ABORT(COMM, ERRORCODE, IERROR)
32
 INTEGER COMM, ERRORCODE, IERROR
33
34
 This routine makes a "best attempt" to abort all tasks in the group of comm. This
35
 function does not require that the invoking environment take any action with the error
36
 code. However, a Unix or POSIX environment should handle this as a return errorcode
37
 from the main program.
38
 It may not be possible for an MPI implementation to abort only the processes repre-
39
 sented by comm if this is a subset of the processes. In this case, the MPI implementation
40
 should attempt to abort all the connected processes but should not abort any unconnected
41
 processes. If no processes were spawned, accepted, or connected then this has the effect of
42
 aborting all the processes associated with MPI_COMM_WORLD.
43
44
 Advice to implementors.
 After aborting a subset of processes, a high quality im-
45
 plementation should be able to provide error handling for communicators, windows,
46
 and files involving both aborted and non-aborted processes. As an example, if the
47
 user changes the error handler for MPI_COMM_WORLD to MPI_ERRORS_RETURN or a
48
 custom error handler, when a subset of MPI_COMM_WORLD is aborted, the remaining
```

processes in MPI_COMM_WORLD should be able to continue communicating with each other and receive appropriate error codes when attempting communication with an aborted process. (End of advice to implementors.)

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 20to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that 21they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then 22no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts 23of 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 true if MPI was finalized (logical) flag

int MPI_Finalized(int *flag)

#### **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

 24 

25

2627

28

29

30

 31 

32

33

34 35

36

37

38

39

40

41

4243 44

45

1	MPI_Finalized(flag, ierror)
2	LOGICAL, INTENT(OUT) :: flag
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	
5	MPI_FINALIZED(FLAG, IERROR)
6	LOGICAL FLAG
7	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. This function must always be thread-safe, as defined in Section 12.4.

- 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.)
- 16 17 18

19 20

21

8

9

10

11

12

13 14

15

#### 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

22 23 24

#### 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.

40 It is suggested that

 $41 \\ 42$ 

46

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.

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

#### Unofficial Draft for Comment Only

order that mpiexec be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 10.3.4).

Analogous to MPI_COMM_SPAWN, we have

mpiexec -n <		naxproc	s>
-soft	<		>
-host	<		>
-arch	<		>
-wdir	<		>
-path	<		>
-file	<		>
<comman< td=""><td>nd</td><td>line&gt;</td><td></td></comman<>	nd	line>	

for the case where a single command line for the application program and its arguments will suffice. See Section 10.3.4 for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats: Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the -n x argument is optional; the default is implementation dependent. It might be 1, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the info argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

mpiexec -configfile <filename>

where the lines of *<*filename> are of the form separated by the colons in Form A. Lines beginning with '#' are comments, and lines may be continued by terminating the partial line with ' $\$ '.

**Example 8.11** Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

**Example 8.12** Start 10 processes on the machine called ferrari:

mpiexec -n 10 -host ferrari myprog

 $\mathbf{2}$ 

1 2	<b>Example 8.13</b> Start three copies of the same program with different command-line arguments:
3	
4	<pre>mpiexec myprog infile1 : myprog infile2 : myprog infile3</pre>
5	
6	Exemple 8.14 Start the second program on five Sung and the stress program on 10
7 8	<b>Example 8.14</b> Start the ocean program on five Suns and the atmos program on 10 RS/6000's:
9	
10	mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
11	It is assumed that the implementation in this case has a method for chaosing heats of
12	It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their paper in the order manifold
13	the appropriate type. Their ranks are in the order specified.
14	
15	<b>Example 8.15</b> Start the ocean program on five Suns and the atmos program on 10
16 17	RS/6000's (Form B):
18	mpiexec -configfile myfile
19	mpiexec -configitie myffie
20	where myfile contains
21	
22	-n 5 -arch sun ocean
23	-n 10 -arch rs6000 atmos
24	
25	(End of advice to implementors.)
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. Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

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 any MPI 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

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 rules for how info value strings are converted to other types, but to ensure portability, every  $\mathbf{2}$ implementation must support the following representations. Valid values for a boolean must 3 include the strings "true" and "false" (all lowercase). For integers, valid values must include 4 string representations of decimal values of integers that are within the range of a standard  $\mathbf{5}$ integer type in the program. (However it is possible that not every integer is a valid value 6 for a given key.) On positive numbers, + signs are optional. No space may appear between  $\overline{7}$ a + or - sign and the leading digit of a number. For comma separated lists, the string 8 must contain valid elements separated by commas. Leading and trailing spaces are stripped 9 automatically from the types of info values described above and for each element of a comma 10 separated list. These rules apply to all info values of these types. Implementations are free 11to specify a different interpretation for values of other info keys. 1213 MPI_INFO_CREATE(info) 1415OUT info info object created (handle) 1617int MPI_Info_create(MPI_Info *info) 18 MPI_Info_create(info, ierror) 19TYPE(MPI_Info), INTENT(OUT) :: info 20INTEGER, OPTIONAL, INTENT(OUT) :: 21ierror 22MPI_INFO_CREATE(INFO, IERROR) 23INTEGER INFO, IERROR  24 MPI_INFO_CREATE creates a new info object. The newly created object contains no 2526key/value pairs. 2728MPI_INFO_SET(info, key, value) 29INOUT info info object (handle) 30 31IN key key (string) 32 IN value value (string) 33 34int MPI_Info_set(MPI_Info info, const char *key, const char *value) 35 36 MPI_Info_set(info, key, value, ierror) 37 TYPE(MPI_Info), INTENT(IN) :: info 38 CHARACTER(LEN=*), INTENT(IN) :: key, value 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40MPI_INFO_SET(INFO, KEY, VALUE, IERROR) 41 INTEGER INFO, IERROR 42CHARACTER*(*) KEY, VALUE 43 44MPI_INFO_SET adds the (key,value) pair to info, and overrides the value if a value for 45

the same key was previously set. key and value are null-terminated strings in C. In Fortran, leading and trailing spaces in key and value are stripped. If either key or value are larger than the allowed maximums, the errors MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are

1 raised, respectively. 2 3 MPI_INFO_DELETE(info, key) 4 5INOUT info info object (handle) 6 IN key key (string) 7 8 int MPI_Info_delete(MPI_Info info, const char *key) 9 10 MPI_Info_delete(info, key, ierror) 11 TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=*), INTENT(IN) :: key 1213 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14MPI_INFO_DELETE(INFO, KEY, IERROR) 15INTEGER INFO, IERROR 16CHARACTER*(*) KEY 1718 MPI_INFO_DELETE deletes a (key,value) pair from info. If key is not defined in info, 19 the call raises an error of class MPI_ERR_INFO_NOKEY. 2021MPI_INFO_GET(info, key, valuelen, value, flag) 2223IN info info object (handle)  24 IN key key (string) 25IN valuelen length of value arg (integer) 2627OUT value value (string) 28OUT flag true if key defined, false if not (boolean) 29 30 int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value, 31int *flag) 32 33 MPI_Info_get(info, key, valuelen, value, flag, ierror) 34 TYPE(MPI_Info), INTENT(IN) :: info 35CHARACTER(LEN=*), INTENT(IN) :: key 36 INTEGER, INTENT(IN) :: valuelen 37 CHARACTER(LEN=valuelen), INTENT(OUT) :: value 38 LOGICAL, INTENT(OUT) :: flag 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR) 41 INTEGER INFO, VALUELEN, IERROR 42CHARACTER*(*) KEY, VALUE 43 LOGICAL FLAG 4445This function retrieves the value associated with key in a previous call to 46MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value, 47

**Unofficial Draft for Comment Only** 

otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters

```
1
 available in value. If it is less than the actual size of the value, the value is truncated. In
\mathbf{2}
 C, valuelen should be one less than the amount of allocated space to allow for the null
3
 terminator.
4
 If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
5
6
 MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
7
8
 IN
 info
 info object (handle)
9
 IN
 key (string)
 key
10
 OUT
 valuelen
 length of value arg (integer)
11
12
 OUT
 flag
 true if key defined, false if not (boolean)
13
14
 int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
15
 int *flag)
16
 MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
17
 TYPE(MPI_Info), INTENT(IN) :: info
18
 CHARACTER(LEN=*), INTENT(IN) ::
 kev
19
 INTEGER, INTENT(OUT) :: valuelen
20
 LOGICAL, INTENT(OUT) :: flag
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
^{24}
 INTEGER INFO, VALUELEN, IERROR
25
 LOGICAL FLAG
26
 CHARACTER*(*) KEY
27
 Retrieves the length of the value associated with key. If key is defined, valuelen is set to
28
 the length of its associated value and flag is set to true. If key is not defined, valuelen is not
29
 touched and flag is set to false. The length returned in C does not include the end-of-string
30
 character.
^{31}
 If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
32
33
34
 MPI_INFO_GET_NKEYS(info, nkeys)
35
 IN
 info
 info object (handle)
36
37
 OUT
 nkeys
 number of defined keys (integer)
38
39
 int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
40
41
 MPI_Info_get_nkeys(info, nkeys, ierror)
42
 TYPE(MPI_Info), INTENT(IN) :: info
 INTEGER, INTENT(OUT) :: nkeys
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
 MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
46
 INTEGER INFO, NKEYS, IERROR
47
48
 MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
```

MPI_INFO_GET_NTHKEY(info, n, key) ¹			1
IN	info	info object (handle)	2 3
IN	n	key number (integer)	4
OUT	key	key (string)	5
			6
int MPI_	Info_get_nthkey(MPI_Info	info, int n, char *key)	7 8
MPI_Info	_get_nthkey(info, n, key,	ierror)	9
TYPE	(MPI_Info), INTENT(IN) ::	info	10
	GER, INTENT(IN) :: n		11
	ACTER(LEN=*), INTENT(OUT)	•	12
	GER, OPTIONAL, INTENT(OUT		13 14
	_GET_NTHKEY(INFO, N, KEY,	IERROR)	15
	GER INFO, N, IERROR ACTER*(*) KEY		16
			17
		d key in info. Keys are numbered $0 \dots N-1$ where	18
	0	GET_NKEYS. All keys between 0 and $N-1$ are	19
-	with MPI_INFO_SET or MPI_I	of a given key does not change as long as info is not NEO DELETE	20 21
mounicu			22
			23
MPI_INF(	DUP(info, newinfo)		24
IN	info	info object (handle)	25
OUT	newinfo	info object (handle)	26
			27 28
int MPI_	Info_dup(MPI_Info info, M	PI_Info *newinfo)	29
MPI_Info	_dup(info, newinfo, ierro:	r)	30
TYPE	(MPI_Info), INTENT(IN) ::	info	31
TYPE(MPI_Info), INTENT(OUT) :: newinfo		32	
INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror	33
MPI_INFO	_DUP(INFO, NEWINFO, IERRO	R)	34 35
INTE	GER INFO, NEWINFO, IERROR		36
MPI_	INFO_DUP duplicates an exis	sting info object, creating a new object, with the	37
	,value) pairs and the same ord		38
			39
MPI INFO	D_FREE(info)		40
		info object (handlo)	41 42
INOUT	info	info object (handle)	43
int MPT	<pre>Info_free(MPI_Info *info)</pre>		44
			45
MPI_Info_free(info, ierror)			46
	(MPI_Info), INTENT(INOUT) GER, OPTIONAL, INTENT(OUT)		47
		, Terror	48

Unofficial Draft for Comment Only

12	MPI_INFO_FREE(INFO, IERROR) INTEGER INFO, IERROR
3 4 5	This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is interpreted each time the info is passed to a routine. Changes to an info after return from
6	a routine do not affect that interpretation.
7 8	
9	
10	
11	
12	
13	
14 15	
16	
17	
18	
19	
20	
21 22	
23	
24	
25	
26	
27	
28 29	
30	
31	
32	
33	
34 35	
36	
37	
38	
39	
40	
41 42	
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 allows for a variety of approaches to process management while placing minimal restrictions on the execution environment.

The MPI model for process creation allows both the creation of an initial set of processes related by their membership in a common MPI_COMM_WORLD and the creation and management of processes after an MPI application has been started. A major impetus for the latter form of process creation comes from the PVM [24] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. MPI assumes that resource control is provided externally — probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for including process management in MPI are both technical and practical. Important classes of message-passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM's capabilities for process and resource management. The lack of these features would be a practical stumbling block to migration.

The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.
- MPI must not take over operating system responsibilities. It should instead provide a

 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 37 that there is an enormous range of runtime environments and application requirements, and 38 39 MPI must not be tailored to any particular one. Examples of such environments are:

40 41

42

43

44

45

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

1

2

3

4 5

> 6 7

8

9

10

11

12

13

1415

16

- Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a "virtual machine" out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.
- Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.
- Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API. An example of such an API would be the PVM task and machine management routines — pvm_addhosts, pvm_config, pvm_tasks, etc., possibly modified to return an MPI (group, rank) when possible. A Condor or PBS API would be another possibility.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of environment-specific 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.

#### **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

• An attribute MPI_UNIVERSE_SIZE (See Section $10.5.1$ ) on MPI_COMM_WORLD tells a
program how "large" the initial runtime environment is, namely how many processes
can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from
this value to find out how many processes might usefully be started in addition to
those already running.

### 10.3 Process Manager Interface

10.3.1 Processes in MPI

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.

 $14 \\ 15$ 

19

20

21

22

23

8 9

10

### 10.3.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

Advice to users. It is possible in MPI to start a static SPMD or MPMD application by first starting one process and having that process start its siblings with MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

24 25 26

27

28

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)

29 30 31	IN	command	name of program to be spawned (string, significant only at root)
31 32 33	IN	argv	arguments to <b>command</b> (array of strings, significant only at root)
34 35	IN	maxprocs	maximum number of processes to start (integer, sig- nificant only at root)
36 37 38 39	IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, signifi- cant only at root)
40 41	IN	root	rank of process in which previous arguments are examined (integer)
42 43	IN	comm	intracommunicator containing group of spawning processes (handle)
44 45 46	OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
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,
 MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
 2
 3
 int array_of_errcodes[])
 4
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
 5
 array_of_errcodes, ierror)
 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

33 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.

```
Example 10.1 Examples of argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
```

```
MPI_Comm_spawn(command, argv, ...);
```

or, if not everything is known at compile time:

```
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 for more information on the soft key for info.

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N," you should do a soft spawn, where the set of allowed values  $\{m_i\}$  is  $\{0...N\}$ . However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the mpi_f08 module and INTEGER in Fortran with the mpi module or the include file mpif.h. It is a container for a

### Unofficial Draft for Comment Only

 24 

 31 

number of user-specified (key,value) pairs. key and value are strings (null-terminated char*
 in C, character*(*) in Fortran). Routines to create and manipulate the info argument are described in Chapter 9.

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.

⁸ MPI does not specify the content of the info argument, except to reserve a number of ⁹ special key values (see Section 10.3.4). The info argument is quite flexible and could even ¹⁰ be used, for example, to specify the executable and its command-line arguments. In this ¹¹ case the command argument to MPI_COMM_SPAWN could be empty. The ability to do this ¹² follows from the fact that MPI does not specify how an executable is found, and the info ¹³ argument can tell the runtime system where to "find" the executable "" (empty string). Of ¹⁴ 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

 31 

32

33 34 35

43

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,  $array_of_errcodes$  is filled in with the value MPI_SUCCESS. If only m23 $(0 \le m \le maxprocs)$  processes are spawned, m of the entries will contain MPI_SUCCESS and  24 the rest will contain an implementation-specific error code indicating the reason MPI could 25not start the process. MPI does not specify which entries correspond to failed processes. 26An implementation may, for instance, fill in error codes in one-to-one correspondence with 27a detailed specification in the info argument. These error codes all belong to the error class 28MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application 29may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. 30

- Advice to implementors. MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4. (End of advice to implementors.)
- ³⁶ MPI_COMM_GET_PARENT(parent)

parent

```
<sup>37</sup>
<sub>38</sub> OUT
```

the parent communicator (handle)

```
39
40 int MPI_Comm_get_parent(MPI_Comm *parent)
```

```
 MPI_Comm_get_parent(parent, ierror)
 TYPE(MPI_Comm) INTENT(OUT) ·· pa
```

```
TYPE(MPI_Comm), INTENT(OUT) :: parent
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
44
45 MPI_COMM_GET_PARENT(PARENT, IERROR)
```

```
45
46 INTEGER PARENT, IERROR
```

⁴⁷ If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE,
 ⁴⁸ 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.

 $\mathbf{2}$ 

 $\overline{7}$ 

#### 1MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, $\mathbf{2}$ array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes) 3 4 IN count number of commands (positive integer, significant to 5MPI only at root — see advice to users) 6 IN array_of_commands programs to be executed (array of strings, significant 7 only at root) 8 9 IN array_of_argv arguments for commands (array of array of strings, 10 significant only at root) 11 IN array_of_maxprocs maximum number of processes to start for each com-12mand (array of integer, significant only at root) 13 IN array_of_info info objects telling the runtime system where and how 14to start processes (array of handles, significant only at 15root) 1617 IN rank of process in which previous arguments are exroot 18 amined (integer) 19 IN intracommunicator containing group of spawning procomm 20cesses (handle) 21OUT intercomm intercommunicator between original group and newly 22spawned group (handle) 2324OUT array_of_errcodes one error code per process (array of integer) 2526int MPI_Comm_spawn_multiple(int count, char *array_of_commands[], 27char **array_of_argv[], const int array_of_maxprocs[], 28const MPI_Info array_of_info[], int root, MPI_Comm comm, 29 MPI_Comm *intercomm, int array_of_errcodes[]) 30 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv, 31 array_of_maxprocs, array_of_info, root, comm, intercomm, 32 array_of_errcodes, ierror) 33 INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root 34 CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*) 35 CHARACTER(LEN=*), INTENT(IN) :: array_of_argv(count, *) 36 TYPE(MPI_Info), INTENT(IN) :: array_of_info(*) 37 TYPE(MPI_Comm), INTENT(IN) :: comm 38 TYPE(MPI_Comm), INTENT(OUT) :: intercomm 39 INTEGER :: array_of_errcodes(*) 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV, 43 ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM, 44ARRAY_OF_ERRCODES, IERROR) 45INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM, 46INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR 47 CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *) 48

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the j-th argument to command number i.

*Rationale.* This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv must be the same as count. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI_COMM_SPAWN_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI_ARGVS_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (End of rationale.)

Advice to users. The argument count is interpreted by MPI only at the root, as is array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array_of_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (End of advice to users.)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be (char ***)0 in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first element is null ((char *)0 in C and empty string in Fortran). In Fortran at non-root processes, the count argument must be set to a value that is consistent with the provided array_of_argv although the content of these arguments has no meaning for this operation.

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the commands are specified in MPI_COMM_SPAWN_MULTIPLE. Assume that  $m_1$  processes are generated by the first command,  $m_2$  by the second, etc. The processes corresponding to the first command have ranks  $0, 1, \ldots, m_1-1$ . The processes in the second command have ranks  $m_1, m_1+1, \ldots, m_1+m_2-1$ . The processes in the third have ranks  $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 - m_3 - 1$ , etc.

Advice to users. Calling MPI_COMM_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDs whereas

MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, so the two methods are not completely equivalent. There are also two performancerelated reasons why, if you need to spawn multiple executables, you may want to use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (End of advice to users.)

### 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
\overline{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
 call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
29
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 formation of proto, we mean.	9
1. a means $a$	10
2. a:b means $a, a + 1, a + 2,, b$	11
3. a:b:c means $a, a + c, a + 2c, \ldots, a + ck$ , where for $c > 0$ , k is the largest integer	12
for which $a + ck \le b$ and for $c < 0$ , k is the largest integer for which $a + ck \ge b$ .	13
If $b > a$ then c must be positive. If $b < a$ then c must be negative.	14 15
Examples:	16
1. a: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 a power-of-two num-	20
ber of processes.	21
-	22
4. 2:10000:2 allows an even number of processes.	23
5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.	24
	25
10.3.5 Spawn Example	26
Manager-worker Example Using MPI_COMM_SPAWN	27 28
/* manager */	29
<pre>#include "mpi.h"</pre>	30
int main(int argc, char *argv[])	31
{	32
<pre>int world_size, universe_size, *universe_sizep, flag;</pre>	33
MPI_Comm everyone; /* intercommunicator */	34
char worker_program[100];	35
	36
<pre>MPI_Init(&amp;argc, &amp;argv);</pre>	37
<pre>MPI_Comm_size(MPI_COMM_WORLD, &amp;world_size);</pre>	38
	39
<pre>if (world_size != 1) error("Top heavy with management");</pre>	40
	41 42
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,	43
<pre>&amp;universe_sizep, &amp;flag); if (!flag) {</pre>	44
printf("This MPI does not support UNIVERSE_SIZE. How many\n\	45
processes total?");	46
-	47
<pre>scanf("%d", &amp;universe_size);</pre>	47

} else universe_size = *universe_sizep;

Unofficial Draft for Comment Only

 $1 \\ 2$ 

3

4

5

6 7

8

```
1
 if (universe_size == 1) error("No room to start workers");
2
3
 /*
4
 * Now spawn the workers. Note that there is a run-time determination
5
 * of what type of worker to spawn, and presumably this calculation must
6
 * be done at run time and cannot be calculated before starting
7
 * the program. If everything is known when the application is
8
 * first started, it is generally better to start them all at once
9
 * in a single MPI_COMM_WORLD.
10
 */
11
12
 choose_worker_program(worker_program);
13
 MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
14
 MPI_INFO_NULL, 0, MPI_COMM_SELF, & everyone,
15
 MPI_ERRCODES_IGNORE);
16
 /*
17
 * Parallel code here. The communicator "everyone" can be used
18
 * to communicate with the spawned processes, which have ranks 0,...
19
 * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
20
 * "everyone".
21
 */
22
23
 MPI_Finalize();
^{24}
 return 0;
25
 }
26
 /* worker */
27
28
 #include "mpi.h"
29
 int main(int argc, char *argv[])
30
^{31}
 ſ
32
 int size;
33
 MPI_Comm parent;
34
 MPI_Init(&argc, &argv);
 MPI_Comm_get_parent(&parent);
35
 if (parent == MPI_COMM_NULL) error("No parent!");
36
 MPI_Comm_remote_size(parent, &size);
37
 if (size != 1) error("Something's wrong with the parent");
38
39
 /*
40
41
 * Parallel code here.
 \ast The manager is represented as the process with rank 0 in (the remote
42
 * group of) the parent communicator. If the workers need to communicate
43
 * among themselves, they can use MPI_COMM_WORLD.
44
 */
45
46
47
 MPI_Finalize();
48
 return 0;
```

#### }

### 10.4 Establishing Communication

This section provides functions that establish communication between two sets of MPI processes that do not share a communicator.

Some situations in which these functions are useful are:

- 1. Two parts of an application that are started independently need to communicate.
- 2. A visualization tool wants to attach to a running process.
- 3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI intercommunicator, where the two groups of the intercommunicator are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client/server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that does not participate in a collective operation may cause a server to crash or hang. (*End of advice to users.*)

### 10.4.1 Names, Addresses, Ports, and All That

Almost all of the complexity in MPI client/server routines addresses the question "how does the client find out how to contact the server?" The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication.

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client does not really care what server it contacts, only that it be able to get in touch with one that can handle its request.

Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple, portable code. The following should be compatible with MPI:

- The server resides at a well-known internet address host:port.
- The server prints out an address to the terminal; the user gives this address to the client program.

#### Unofficial Draft for Comment Only

 24 

 $\frac{44}{45}$ 

1 2

3

4

5 6

7

8

9

10

11

12

13

25

30

31

32

33

36

37

38

• The server places the address information on a nameserver, 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.
- ³⁴₃₅ 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.

- 39 40 41
- MPI_OPEN_PORT(info, port_name)

42	IN	info	implementation-specific information on how to estab-
43			lish an address (handle)
44	OUT	port_name	newly established port (string)
45			
46	int MDT (	Doon port (MDT Info	info char that name)
47	IIIC MFI_(	pheu-hor c(ub1-1010	info, char *port_name)
48	MDT Onen	port(info port na	me ierror)

MPI_Open_port(info, port_name, ierror)

```
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.

Unofficial Draft for Comment Only

 24 

```
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)
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
```

IN	port_name	network address (string, used only on root)	2
IN	info	implementation-dependent information (handle, used only on root)	3 4 5
IN	root	rank in <b>comm</b> of root node (integer)	6
IN	comm	intracommunicator over which call is collective (han- dle)	7 8
OUT	newcomm	intercommunicator with server as remote group (han- dle)	9 10 11
			12
int MPI_	Comm_connect(const char *	<pre>port_name, MPI_Info info, int root,</pre>	13
	MPI_Comm comm, MPI_C	-	14
MDT Comm	connect(nort name info	reat comm nouccomm iarrar)	15
	ACTER(LEN=*), INTENT(IN)	root, comm, newcomm, ierror)	16
	C(MPI_Info), INTENT(IN) ::	-	17
	GER, INTENT(IN) :: root	1110	18
	C(MPI_Comm), INTENT(IN) ::	Comm	19
	C(MPI_Comm), INTENT(OUT) :		20
	GER, OPTIONAL, INTENT(OUT		21
			22
		ROOT, COMM, NEWCOMM, IERROR)	23
	ACTER*(*) PORT_NAME	2010/ TERDOD	24
TNLF	GER INFO, ROOT, COMM, NEW	CUMM, IEKKUK	25
This	routine establishes communic	ation with a server specified by port_name. It is	26 27
collective	area the colling communicate	n and naturna an intercommunicator in which the	41

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 8 is, the domain over which the service_name can be retrieved. If the domain is the empty 9 set, that is, if no client can retrieve the information, then we say that name publishing 10 is not supported. Implementations should document how the scope is determined. High-11 quality implementations will give some control to users through the info arguments to name 12publishing functions. Examples are given in the descriptions of individual functions. 13

14

```
15
16
```

33

MPI_PUBLISH_NAME(service_name, info, port_name)

17	IN	service_name	a service name to associate with the port (string)
18	IN	info	implementation-specific information (handle)
19 20	IN	port_name	a port name (string)
21 22 23	int MPI_1	Publish_name(const char *: const char *port_nam	service_name, MPI_Info info, e)
24 25 26 27 28	TYPE CHAR	ish_name(service_name, int (MPI_Info), INTENT(IN) :: ACTER(LEN=*), INTENT(IN) GER, OPTIONAL, INTENT(OUT)	info :: service_name, port_name
29 30 31	INTE	ISH_NAME(SERVICE_NAME, IN GER INFO, IERROR ACTER*(*) SERVICE_NAME, P(	
32			

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.

⁴⁰ MPI permits publishing more than one service_name for a single port_name. On the ⁴¹ other hand, if service_name has already been published within the scope determined by info, ⁴² the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through ⁴³ a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple ⁴⁴ servers with the same service in the same scope. In this case, an implementation-defined ⁴⁵ policy will determine which of several port names is returned by MPI_LOOKUP_NAME.

⁴⁶ Note that while service_name has a limited scope, determined by the implementation,
 ⁴⁷ 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.*)

		lee_name, into, port_name)
IN	service_name	a service name (string)
IN	info	implementation-specific information (handle)
IN	port_name	a port name (string)

MPLUNPUBLISH NAME(service name info port name)

```
MPI_Unpublish_name(service_name, info, port_name, ierror)
 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

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

 $\mathbf{2}$ 

1 MPI_LOOKUP_NAME(service_name, info, port_name) 2 IN service_name a service name (string) 3 IN info implementation-specific information (handle) 4 5OUT port_name a port name (string) 6  $\overline{7}$ int MPI_Lookup_name(const char *service_name, MPI_Info info, 8 char *port_name) 9 MPI_Lookup_name(service_name, info, port_name, ierror) 10 CHARACTER(LEN=*), INTENT(IN) :: service_name 11 TYPE(MPI_Info), INTENT(IN) :: info 12CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) 16CHARACTER*(*) SERVICE_NAME, PORT_NAME 17 INTEGER INFO, IERROR 18 This function retrieves a port_name published by MPI_PUBLISH_NAME with 19service_name. If service_name has not been published, it raises an error in the error class 20MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the 21largest possible port name (see discussion above under MPI_OPEN_PORT). 22 If an implementation allows multiple entries with the same service_name within the 23same scope, a particular **port_name** is chosen in a way determined by the implementation.  24 If the info argument was used with MPI_PUBLISH_NAME to tell the implementation 2526how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME. 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;

/* */	1
MPI_Open_port(MPI_INFO_NULL, myport);	2
<pre>printf("port name is: %s\n", myport);</pre>	3
	4 5
<pre>MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &amp;intercomm); /* do something with intercomm */</pre>	6
, and the second s	7
The server prints out the port name to the terminal and the user must type it in when	8
starting up the client (assuming the MPI implementation supports stdin such that this works). On the client side:	9 10
	11
MPI_Comm intercomm;	12
<pre>char name[MPI_MAX_PORT_NAME]; printf("enter port name: ");</pre>	13
<pre>gets(name);</pre>	14
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);	15
	16
Ocean/Atmosphere — Relies on Name Publishing	17
,	18 19
In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere	19 20
climate model. It assumes that the MPI implementation publishes names.	20 21
	21
<pre>MPI_Open_port(MPI_INFO_NULL, port_name);</pre>	23
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);	24
	25
<pre>MPI_Comm_accept(port_name, MPI_INF0_NULL, 0, MPI_COMM_SELF, &amp;intercomm);</pre>	26
<pre>/* do something with intercomm */</pre>	27
<pre>MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);</pre>	28
	29
On the direct side.	30
On the client side:	31
<pre>MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);</pre>	32
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,	33 34
&intercomm);	35
	36
Simple Client-Server Example	37
This is a simple example; the server accepts only a single connection at a time and serves	38
that connection until the client requests to be disconnected. The server is a single process.	39
Here is the server. It accepts a single connection and then processes data until it	40
receives a message with tag 1. A message with tag 0 tells the server to exit.	41
	42
<pre>#include "mpi.h"</pre>	43
int main(int argc, char *argv[])	44
{ MPI Comm client:	45
MPI_Comm client; MPI_Status status;	46
char port_name[MPI_MAX_PORT_NAME];	47
enar pertinducin rinkar on intraktion,	48

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.)
- 38 10

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 do not share the same MPI_COMM_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.

### Unofficial Draft for Comment Only

 $\overline{7}$ 

 24 

 $44 \\ 45$ 

1	The following additional rules apply to MPI routines in other chapters:
2 3	• MPI_FINALIZE is collective over a set of connected processes.
4 5 6 7 8	• MPI_ABORT does not abort independent processes. It may abort all processes in the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.
9 10 11 12	• If a process terminates without calling MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.
13	MPI_COMM_DISCONNECT(comm)
14 15 16	INOUT comm communicator (handle)
17	<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>
18 19 20 21	MPI_Comm_disconnect(comm, ierror) 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. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> )
41 42 43 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) MPI_Comm_join(fd, intercomm, ierror) INTEGER, INTENT(IN) :: fd TYPE(MPI_Comm), INTENT(OUT) :: intercomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
MPI_COMM_JOIN(FD, INTERCOMM, IERROR) INTEGER FD, INTERCOMM, IERROR				

MPI_COMM_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [45, 49]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN and should return MPI_COMM_NULL.

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI_COMM_JOIN should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (End of advice to implementors.)

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI_COMM_JOIN. The two processes are referred to as the local and remote processes.

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent (see below).

### 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 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.

⁹ The design of the RMA functions allows implementors to take advantage of fast or ¹⁰ asynchronous communication mechanisms provided by various platforms, such as coherent ¹¹ or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and ¹² communication coprocessors. The most frequently used RMA communication mechanisms ¹³ can be layered on top of message-passing. However, certain RMA functions might need ¹⁴ support for asynchronous communication agents in software (handlers, threads, etc.) in a ¹⁵ distributed memory environment.

¹⁶ We shall denote by **origin** the process that performs the call, and by **target** the ¹⁷ process in which the memory is accessed. Thus, in a put operation, source=origin and ¹⁸ destination=target; in a get operation, source=target and destination=origin.

19 20

21

## 11.2 Initialization

 $^{22}_{23}$  MPI provides the following window initialization functions: MPI_WIN_CREATE,

MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and

²⁴ MPI_WIN_CREATE_DYNAMIC, which are collective on an intracommunicator.

²⁵ MPI_WIN_CREATE allows each process to specify a "window" in its memory that is made ²⁶ accessible to accesses by remote processes. The call returns an opaque object that represents ²⁷ the group of processes that own and access the set of windows, and the attributes of each ²⁸ window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from

²⁵ MPI_WIN_CREATE in that the user does not pass allocated memory;

³¹ MPI_WIN_ALLOCATE returns a pointer to memory allocated by the MPI implementation. ³¹ MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated ³² memory can be accessed from all processes in the window's group with direct load/store ³³ instructions. Some restrictions may apply to the specified communicator.

³⁴ MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control ³⁵ which memory is exposed by the window.

- 38
- 39
- 40 41
- 42

- 44
- 45
- 46
- 47

### 11.2.1 Window Creation

11.2.1	Window Creation		1
			2
			3
MPI_W	/IN_CREATE(base, size, disp_unit	, info, comm, win)	4
IN	base	initial address of window (choice)	5 6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	8 9
IN	info	info argument (handle)	10 11
IN	comm	intra-communicator (handle)	12
OUT	win	window object returned by the call (handle)	13
		<b>3 3 ( )</b>	14 15
int MF	int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,		
MPI_Comm comm, MPI_Win *win)			16 17
MPT Wi	MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)		
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: base		
	ITEGER(KIND=MPI_ADDRESS_KIND		20
IN	TEGER, INTENT(IN) :: disp_	unit	21
ΤY	PE(MPI_Info), INTENT(IN) ::	info	22
ΤY	<pre>PE(MPI_Comm), INTENT(IN) ::</pre>	comm	23
ΤY	<pre>PE(MPI_Win), INTENT(OUT) ::</pre>	win	24
IN	NTEGER, OPTIONAL, INTENT(OUT	') :: ierror	25
MDT WIT	IN CREATE (BASE STZE DISD II	NIT, INFO, COMM, WIN, IERROR)	26
	cype> BASE(*)	NII, INFO, COMA, WIN, ILMUON)	27
	TEGER(KIND=MPI_ADDRESS_KIND	) SIZE	28
	TEGER DISP_UNIT, INFO, COMM		29
			30
Tl	his is a collective call executed b	by all processes in the group of <b>comm</b> . It returns	31

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 18.1.12). 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, rather than a basic integer type, to allow windows that span more memory than can be described with a basic integer type. (End of rationale.)

Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax) 47sizeof (type), for a window that consists of an array of elements of type type. The 48

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

44

4546

32 33

34

35

36 37

38 39

	430	CHAP	TER 11.	ONE-SIDED COMMUNICATIONS
1 2 3 4			-	in RMA calls, and have those scaled geneous environment. ( <i>End of advice</i>
5 6 7		ument provides optimization ndow. The following info ke		he runtime about the expected usage edefined:
8 9 10 11	nization (i.e window. T	e., MPI_WIN_LOCK, MPI_W his implies that this window	VIN_LOCK w is not u	(_ALL) will not be used on the given ased for 3-party communication, and ronous agent activity at this process.
12 13 14		ng — controls the ordering 7.2 for details.	; of accum	nulate operations at the target. See
15 16 17 18 19 20	accumulate same_op_no calls to the eliminate t	calls to the same target a op, then the implementation same target address will us	ddress will on will ass e the sam or certain	tion will assume that all concurrent Il use the same operation. If set to sume that all concurrent accumulate e operation or MPI_NO_OP. This can operation types where the hardware _no_op.
21 22 23 24		all processes, and that all		have provided this info key with the
25 26 27 28	disp_unit is	,		tion may assume that the argument all processes have provided this info
29 30 31 32 33	to query th It is recom	e specified info arguments fo	or window	scribed in Section 11.2.7 can be used as that have been passed to a library. I info keys for each passed window.
34 35 36 37 38 39	windows, in local put and accumu should pose no p associated with a	tion, size, displacement unit ate accesses to a particular roblem. The same area in n	ts, and inf r process nemory m owever, co	y specify completely different target to arguments. As long as all the get, fit their specific target window this ay appear in multiple windows, each ncurrent communications to distinct,
40 41 42 43 44 45 46 47 48	process in a can be a ta ification. F process to implementa	an RMA operation is to per rget of RMA operations and or example, with this defini- use RMA operations, known tion does enforce the specifi	rmit the p d for the i tion, a ser ing that ( fied limits	y that may be accessed from another rogrammer to specify what memory mplementation to enforce that spec- rver process can safely allow a client under the assumption that the MPI on the exposed memory) an error in what was explicitly exposed. ( <i>End of</i>

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) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

Implementors should document any performance impact of window alignment. (*End of advice to implementors.*)

### 11.2.2 Window That Allocates Memory

IN	size	size of window in bytes (non-negative integer)	28
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	29 30 31
IN	info	info argument (handle)	32
IN	comm	intra-communicator (handle)	33
OUT	baseptr	initial address of window (choice)	34 35
OUT	win	window object returned by the call (handle)	36
			37
- MDT	Vin allegate (MDT Aint dia	a int dian unit MDI Infa infa	38

MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
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

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

39

40

41

42

43

44

45

46

47

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
3	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
5	
6	This is a collective call executed by all processes in the group of comm. On each
7	process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
8 9	window object that can be used by all processes in comm to perform RMA operations. The
10	returned memory consists of size bytes local to each process, starting at address baseptr
11	and is associated with the window as if the user called MPI_WIN_CREATE on existing memory. The size argument may be different at each process and $size = 0$ is valid; however, a
12	library might allocate and expose more memory in order to create a fast, globally symmetric
13	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
14	Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2
15	for an explanation of the type used for <b>baseptr</b> .
16	If the Fortran compiler provides TYPE (C_PTR), then the following generic interface must
17	be provided in the mpi module and should be provided in mpif.h through overloading,
18	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
19	BASEPTR, but with a different specific procedure name:
20	INTERFACE MPI_WIN_ALLOCATE
21 22	SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
22	WIN, IERROR)
24	IMPORT :: MPI_ADDRESS_KIND
25	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
26	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
27	END SUBROUTINE
28	SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
29	WIN, IERROR)
30	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
31	IMPORT :: MPI_ADDRESS_KIND
32	INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
33 34	TYPE(C_PTR) :: BASEPTR
35	END SUBROUTINE
36	END INTERFACE
37	
38	The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR.
39	The implied specific procedure names are described in Section 18.1.5.
40	Rationale. By allocating (potentially aligned) memory instead of allowing the user
41	to pass in an arbitrary buffer, this call can improve the performance for systems with
42	remote direct memory access. This also permits the collective allocation of memory
43	and supports what is sometimes called the "symmetric allocation" model that can be
44	more scalable (for example, the implementation can arrange to return an address for
45 46	the allocated memory that is the same on all processes). (End of rationale.)
40 47	The inferrence on the used to specify kints similar to the inferrence for
48	The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

# Unofficial Draft for Comment Only

#### 11.2.3 Window That Allocates Shared Memory

MPI_WI	N_ALLOCATE_SHARE	D(size, disp_unit, info, comm, baseptr, win)	4 5
IN	size	size of local window in bytes (non-negative integer)	6
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	7 8
IN	info	info argument (handle)	9
IN	comm	intra-communicator (handle)	10 11
OUT	baseptr	address of local allocated window segment (choice)	12
OUT	win	window object returned by the call (handle)	13
			14
int MPI	_Win_allocate_shared	d(MPI_Aint size, int disp_unit, MPI_Info info,	15 16
	MPI_Comm comm	, void *baseptr, MPI_Win *win)	16
MPI Win	allocate shared(siz	ze, disp_unit, info, comm, baseptr, win, ierror)	18
		_C_BINDING, ONLY : C_PTR	19
		SS_KIND), INTENT(IN) :: size	20
INT	EGER, INTENT(IN) ::	disp_unit	21
TYP	E(MPI_Info), INTENT	(IN) :: info	22
TYP	E(MPI_Comm), INTENT	(IN) :: comm	23
TYP	TYPE(C_PTR), INTENT(OUT) :: baseptr		
TYP	E(MPI_Win), INTENT((	DUT) :: win	25
INT	EGER, OPTIONAL, INT	ENT(OUT) :: ierror	26
мрт цтм		ZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	27
_	·	D, COMM, WIN, IERROR	28
		SS_KIND) SIZE, BASEPTR	29
TWT	EGEN(KIND-FIFI_ADDRE)	JO_NIMU/ DIZE, DROEFIR	30

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 that is shared among all processes in comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used for load/store accesses on the calling process. The locally allocated memory can be the target of load/store accesses by remote processes; the base pointers for other processes can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a window object that can be used by all processes in comm to perform RMA operations. The size argument may be different at each process and size = 0 is valid. It is the user's responsibility to ensure that the communicator comm represents a group of processes that can create a shared memory segment that can be accessed by all processes in the group. The discussions of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 8.2 also apply to MPI_WIN_ALLOCATE_SHARED; in particular, see the rationale in Section 8.2for an explanation of the type used for **baseptr**. The allocated memory is contiguous across process ranks unless the info key alloc_shared_noncontig is specified. Contiguous across process ranks means that the first address in the memory segment of process i is consecutive with the last address in the memory segment of process i-1. This may enable the user to calculate remote address offsets with local information only.

If the Fortran compiler provides TYPE(C_PTR), then the following generic 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 specific procedure name:

```
6
 INTERFACE MPI_WIN_ALLOCATE_SHARED
 SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
\overline{7}
 BASEPTR, WIN, IERROR)
8
 IMPORT :: MPI_ADDRESS_KIND
9
 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
10
11
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
 END SUBROUTINE
12
 SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
13
 BASEPTR, WIN, IERROR)
14
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
15
 IMPORT :: MPI_ADDRESS_KIND
16
 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
17
 INTEGER(KIND=MPI_ADDRESS_KIND) ::
18
 SIZE
19
 TYPE(C_PTR) :: BASEPTR
 END SUBROUTINE
20
 END INTERFACE
21
22
 The base procedure name of this overloaded function is
23
 MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described
^{24}
 in Section 18.1.5.
25
 The info argument can be used to specify hints similar to the info argument for
26
 MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info
27
 key alloc_shared_noncontig allows the library to optimize the layout of the shared memory
28
 segments in memory.
29
30
 Advice to users. If the info key alloc_shared_noncontig is not set to true, the allocation
31
 strategy is to allocate contiguous memory across process ranks. This may limit the
32
 performance on some architectures because it does not allow the implementation to
33
 modify the data layout (e.g., padding to reduce access latency). (End of advice to
34
 users.)
35
36
 Advice to implementors. If the user sets the info key alloc_shared_noncontig to true,
37
 the implementation can allocate the memory requested by each process in a location
38
 that is close to this process. This can be achieved by padding or allocating memory
39
 in special memory segments. Both techniques may make the address space across
40
 consecutive ranks noncontiguous. (End of advice to implementors.)
41
42
 The consistency of load/store accesses from/to the shared memory as observed by the
43
 user program depends on the architecture. A consistent view can be created in the unified
44
 memory model (see Section 11.4) by utilizing the window synchronization functions (see
45
 Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling
46
 MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows
47
```

in the separate memory model.  48 

1

 $\mathbf{2}$ 

3

4

46

47

48

MPI_WIN	_SHARED_QUERY(win, rank, s	size, disp_unit, baseptr)	1	
IN	win	shared memory window object (handle)	2 3	
IN	rank	rank in the group of window win (non-negative integer) or MPI_PROC_NULL	3 4 5	
OUT	size	size of the window segment (non-negative integer)	6	
OUT	disp_unit	local unit size for displacements, in bytes (positive in- teger)	7 8	
OUT	baseptr	address for load/store access to window segment (choice)	9 10 11	
			12	
int MPI_V	1 0	vin, int rank, MPI_Aint *size,	13	
	<pre>int *disp_unit, void</pre>	*baseptr)	14	
MPI_Win_s	shared_query(win, rank, s:	ize, disp_unit, baseptr, ierror)	15 16	
	INTRINSIC :: ISO_C_BIND		17	
	(MPI_Win), INTENT(IN) :: GER, INTENT(IN) :: rank	Win	18	
		), INTENT(OUT) :: size	19	
INTEGER, INTENT(OUT) :: disp unit				
	(C_PTR), INTENT(OUT) :: 1	-	21 22	
INTEC	SER, OPTIONAL, INTENT(OUT)	) :: ierror	23	
MPI_WIN_S	SHARED_QUERY(WIN, RANK, SI	IZE, DISP_UNIT, BASEPTR, IERROR)	24	
	ER WIN, RANK, DISP_UNIT,		25	
INTEC	GER (KIND=MPI_ADDRESS_KINI	D) SIZE, BASEPTR	26 27	
		ocal address for remote memory segments created	28	
		This function can return different process-local ad-	29	
	* ° °	a different processes. The returned memory can be e constraints defined in Section 11.7. This function	30	
	-	r MPI_WIN_FLAVOR_SHARED. If the passed window	31	
		D, the error MPI_ERR_RMA_FLAVOR is raised. When	32 33	
	, <b>i</b> ,	p_unit, and size returned are the pointer, disp_unit,	34	
			35	
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.			36	
If the Fortran compiler provides TYPE (C PTR), then the following generic interface must			37	
		ould be provided in mpif.h through overloading,	38 30	
-	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)			
BASEPTR, 1	out with a different specific pr	ocedure name:	41	
ŢŊŢĘŖĘ∧Ċĭ	E MPI_WIN_SHARED_QUERY		42	
		RY(WIN, RANK, SIZE, DISP_UNIT, &	43	
		BASEPTR, IERROR)	44	
]	IMPORT :: MPI ADDRESS KINI	)	45	

IMPORT :: MPI_ADDRESS_KIND INTEGER WIN, RANK, DISP_UNIT, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

1 END SUBROUTINE 2 SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & 3 BASEPTR, IERROR) 4 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 5IMPORT :: MPI_ADDRESS_KIND 6 INTEGER :: WIN, RANK, DISP_UNIT, IERROR 7 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE 8 TYPE(C_PTR) :: BASEPTR 9 END SUBROUTINE 10 END INTERFACE 11 The base procedure name of this overloaded function is 12MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in 13 Section 18.1.5. 14151611.2.4 Window of Dynamically Attached Memory 17The MPI-2 RMA model requires the user to identify the local memory that may be a 18 target of RMA calls at the time the window is created. This has advantages for both 19 the programmer (only this memory can be updated by one-sided operations and provides 20greater safety) and the MPI implementation (special steps may be taken to make one-21sided access to such memory more efficient). However, consider implementing a modifiable 22 linked list using RMA operations; as new items are added to the list, memory must be 23allocated. In a C or C++ program, this memory is typically allocated using malloc or 24new respectively. In MPI-2 RMA, the programmer must create a window with a predefined 25amount of memory and then implement routines for allocating memory from within the 26window's memory. In addition, there is no easy way to handle the situation where the 27predefined amount of memory turns out to be inadequate. To support this model, the 28routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose 29 memory without remote synchronization. It must be used in combination with the local 30 routines MPI_WIN_ATTACH and MPI_WIN_DETACH.  31 32 33 MPI_WIN_CREATE_DYNAMIC(info, comm, win) 34IN info info argument (handle) 35 36 IN comm intra-communicator (handle) 37 OUT win window object returned by the call (handle) 38 39 int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win) 40 41MPI_Win_create_dynamic(info, comm, win, ierror) 42TYPE(MPI_Info), INTENT(IN) :: info 43 TYPE(MPI_Comm), INTENT(IN) :: comm 44 TYPE(MPI_Win), INTENT(OUT) :: win 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR) 47INTEGER INFO, COMM, WIN, IERROR 48

This 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 described below. This routine returns a window object that can be used by these processes to perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE.

In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for all RMA functions is the address at the target; i.e., the effective window_base is MPI_BOTTOM and the disp_unit is one. For dynamic windows, the target_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI_GET_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI_AINT (see Table 3.3) is able to store addresses from any process. (*End of advice to implementors.*)

Memory at the target cannot be accessed with this window until that memory has been attached using the function MPI_WIN_ATTACH. That is, in addition to using MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use MPI_WIN_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI_WIN_ATTACH(win, base, size)

····· ·_		56, 5126)	35
IN	win	window object (handle)	36
IN	base	initial address of memory to be attached	37
IN	size	size of memory to be attached in bytes	38
	SIZC	size of memory to be attached in bytes	39
			40
int M	IPI_Win_attach(MPI_	Win win, void *base, MPI_Aint size)	41
MPI_W	/in_attach(win, bas	e, size, ierror)	42
1	TYPE(MPI_Win), INTENT(IN) :: win		
Т	YPE(*), DIMENSION(	), ASYNCHRONOUS :: base	44
]	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size		
		INTENT(OUT) :: ierror	46
			47
MPI_W	MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR) 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 19

20

21

22

23

 24 

25

26 27

28

29

30

 31 

1	INTEG	ER WIN, IERROR		
2	<type> BASE(*)</type>			
3	INTEG	ER (KIND=MPI_ADDRESS_KINI	)) SIZE	
4 5	Attack	nes a local memory region beg	ginning at <b>base</b> for remote access within the given	
6	window. T	he memory region specified n	nust not contain any part that is already attached	
7	to the win	dow win, that is, attaching o	verlapping memory concurrently within the same	
8		0	n must be a window that was created with	
9			al memory region attached to the window consists	
10	-		C, base is the starting address of a memory region.	
11			ent of a memory region or a whole array, which y contiguous,' see Section 18.1.12). Multiple (but	
12			e attached to the same window.	
13	non-overiaj	pping) memory regions may b	e attached to the same whitdow.	
14	Ratio	onale. Requiring that mem	ory be explicitly attached before it is exposed to	
15 16	one-s	ided access by other processes	can simplify implementations and improve perfor-	
17		ũ.	y available for RMA operations without requiring a	
18			s needed for some one-sided programming models.	
19	(End	of rationale.)		
20	Advi	ce to users. Attaching mer	mory to a window may require the use of scarce	
21		_	gions of memory is not recommended in portable	
22		, , , , ,	a window may fail if sufficient resources are not	
23		able; this is similar to the beh		
24	The	user is also responsible for en	suring that MPI_WIN_ATTACH at the target has	
25 26		-	s to target that memory with an MPI RMA call.	
27			memory that has not been attached to a window	
28			DYNAMIC is erroneous. ( <i>End of advice to users.</i> )	
29	01000			
30	Advid	ce to implementors. A high	-quality implementation will attempt to make as	
31		0	ing as possible. Any limitations should be docu-	
32	ment	ed by the implementor. ( $End$	of advice to implementors.)	
33	Attack	ing moment is a local enough	ion as defined by MPI, which means that the call	
34			equiring any MPI routine to be called in any other	
35 36		-	the routine MPI_WIN_DETACH. After memory has	
37	been detached, it may not be the target of an MPI RMA operation on that window (unless			
38		y is re-attached with MPI_WI	-	
39			,	
40				
41		DETACH(win, base)		
42	IN	win	window object (handle)	
43	IN	base	initial address of memory to be detached	
44				
45 46	int MPI_W	in_detach(MPI_Win win, co	onst void *base)	
40 47	MPT Win d	etach(win, base, ierror)		
48		MPI_Win), INTENT(IN) ::	win	
	(	,,		

TYPE(*), DIMENSION(), ASYNCHRONOUS :: base	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
	3
MPI_WIN_DETACH(WIN, BASE, IERROR)	4
INTEGER WIN, IERROR	5
<type> BASE(*)</type>	6
Detaches a previously attached memory region beginning at base. The arguments base	7
and win must match the arguments paged to a providus call to MPL WIN ATTACH	8

Advice to users. Detaching memory may permit the implementation to make more efficient use of special memory or provide memory that may be needed by a subsequent MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. (*End of advice to users.*)

and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.

Memory becomes detached when the associated dynamic memory window is freed, see Section 11.2.5.

	11.2.5	Window	Destruction
--	--------	--------	-------------

```
MPI_WIN_FREE(win)
INOUT win window object (handle)
int MPI_Win_free(MPI_Win *win)
MPI_Win_free(win, ierror)
 TYPE(MPI_Win), INTENT(INOUT) :: win
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_FREE(WIN, IERROR)
```

INTEGER WIN, IERROR

Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: e.g., the process has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. The memory associated with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_FREE will free the window memory that was allocated.

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.

```
46
47
48
```

1 MPI_WIN_FREE requires a barrier synchronization: no Advice to implementors. 2 process can return from free until all processes in the group of 3 win call free. This ensures that no process will attempt to access a remote window 4 (e.g., with lock/unlock) after it was freed. The only exception to this rule is when the 5user sets the no_locks info key to true when creating the window. In that case, an MPI 6 implementation may free the local window without barrier synchronization. (End of 7 advice to implementors.) 8 9 11.2.6 Window Attributes 10 The following attributes are cached with a window when the window is created. 11 12MPI_WIN_BASE window base address. 13 MPI_WIN_SIZE window size, in bytes. 14displacement unit associated with the window. MPI_WIN_DISP_UNIT 15MPI_WIN_CREATE_FLAVOR how the window was created. 16memory model for window. MPI_WIN_MODEL 17In C, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), 18 MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag), 19 MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag), 20MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, & create_kind, & flag), and 21MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag) will return in base a 22 pointer to the start of the window win, and will return in size, disp_unit, create_kind, and 23memory_model pointers to the size, displacement unit of the window, the kind of routine 24used to create the window, and the memory model, respectively. A detailed listing of the 25type of the pointer in the attribute value argument to MPI_WIN_GET_ATTR and 26MPI_WIN_SET_ATTR is shown in Table 11.1. 2728Attribute C Type 29 void * MPI_WIN_BASE 30 MPI_WIN_SIZE MPI_Aint *  31 MPI_WIN_DISP_UNIT int * 32 MPI_WIN_CREATE_FLAVOR int * 33 MPI_WIN_MODEL int * 34 35 36 Table 11.1: C types of attribute value argument to MPI_WIN_GET_ATTR and 37 MPI_WIN_SET_ATTR. 38 39 In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror), 40MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror), 41 MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror), 42MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and 43 MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in 44base, size, disp_unit, create_kind, and memory_model the (integer representation of) the 45base address, the size, the displacement unit of the window win, the kind of routine used to 46create the window, and the memory model, respectively. 47The values of create_kind are 48

MPI_WIN_FLAVOR_CREATE	Window was created with MPI_WIN_CREATE.	1
MPI_WIN_FLAVOR_ALLOCATE	Window was created with	2
	MPI_WIN_ALLOCATE.	3
MPI_WIN_FLAVOR_DYNAMIC	Window was created with	4
	MPI_WIN_CREATE_DYNAMIC.	5
MPI_WIN_FLAVOR_SHARED	Window was created with	6
	MPI_WIN_ALLOCATE_SHARED.	7
		8

The values of memory_model are MPI_WIN_SEPARATE and MPI_WIN_UNIFIED. The meaning of these is described in Section 11.4.

In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address is MPI_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 6.7.3.) The value returned for an attribute on a window is constant over the lifetime of the window.

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

MPI_W	IN_GET_GROUP(win, group)		19
IN	win	window object (handle)	20
			21
OUT	group	group of processes which share access to the window $(1, \dots, N)$	22
		(handle)	23
			24
int MP	I_Win_get_group(MPI_Win win	, MPI_Group *group)	25
MDT WH	n_get_group(win, group, ier	ror)	26
	PE(MPI_Win), INTENT(IN) ::		27
	PE(MPI_WIN), INTENT(IN) PE(MPI_Group), INTENT(OUT)		28
	<b>1</b>	0 1	29
ΞN	TEGER, OPTIONAL, INTENT(OUT	) :: lerror	30
MPI_WI	N_GET_GROUP(WIN, GROUP, IER	ROR)	31
IN	TEGER WIN, GROUP, IERROR		32

MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window associated with win. The group is returned in group.

### 11.2.7 Window Info

Hints specified via info (see Section 9) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or use system resources more efficiently. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_WIN_GET_INFO) and that place a restriction on the behavior of the application. 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.

```
1
 Advice to implementors. It may happen that a program is coded with hints for one
\mathbf{2}
 system, and later executes on another system that does not support these hints. In
3
 general, unsupported hints should simply be ignored. Needless to say, no hint can be
4
 mandatory. However, for each hint used by a specific implementation, a default value
5
 must be provided when the user does not specify a value for the hint. (End of advice
6
 to implementors.)
7
8
9
 MPI_WIN_SET_INFO(win, info)
10
11
 INOUT
 window object (handle)
 win
12
 IN
 info
 info object (handle)
13
14
 int MPI_Win_set_info(MPI_Win win, MPI_Info info)
15
16
 MPI_Win_set_info(win, info, ierror)
17
 TYPE(MPI_Win), INTENT(IN) ::
 win
18
 TYPE(MPI_Info), INTENT(IN) ::
 info
19
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
20
 MPI_WIN_SET_INFO(WIN, INFO, IERROR)
21
 INTEGER WIN, INFO, IERROR
22
23
 MPI_WIN_SET_INFO updates the hints of the window associated with win using the
^{24}
 hints provided in info. This operation has no effect on previously set or defaulted hints
25
 that are not specified by info. It also has no effect on previously set or defaulted hints that
26
 are specified by info, but are ignored by the MPI implementation in this call to
27
 MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be
28
 different on each process, but any info entries that an implementation requires to be the
29
 same on all processes must appear with the same value in each process's info object.
30
^{31}
 Some info items that an implementation can use when it creates
 Advice to users.
32
 a window cannot easily be changed once the window has been created. Thus, an
33
 implementation may ignore hints issued in this call that it would have accepted in a
34
 creation call. An implementation may also be unable to update certain info hints in a
 call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether
35
36
 info changes were ignored by the implementation. (End of advice to users.)
37
38
39
 MPI_WIN_GET_INFO(win, info_used)
40
 IN
 window object (handle)
41
 win
42
 OUT
 info_used
 new info object (handle)
43
44
 int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
45
46
 MPI_Win_get_info(win, info_used, ierror)
47
 TYPE(MPI_Win), INTENT(IN) :: win
48
 TYPE(MPI_Info), INTENT(OUT) :: info_used
```

CHAPTER 11. ONE-SIDED COMMUNICATIONS

442

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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 associated with win. The current setting of all hints related to this window is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. 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.

## 11.3 Communication Calls

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET transfer data from the target memory to the caller memory; MPI_ACCUMULATE and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI_GET_ACCUMULATE,

MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.5. Transfers can also be completed with calls to flush routines; see Section 11.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3.

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 resulting data values, or outcome, of concurrent conflicting accesses to the same memory locations is undefined; if a location is updated by a put or accumulate operation, then the outcome of loads or other RMA operations is undefined until the updating operation has completed at the target. There is one exception to this rule; namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred in some order. In addition, the outcome of concurrent load/store and RMA updates to the same memory location is undefined. These restrictions are described in more detail in Section 11.7.

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-

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4 5

6

 $\overline{7}$ 

8

9

10

11

12 13

14 15

16

17

18

19

20

21

22

23

 24 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

	444	CH	HAPTER 11. ONE-SIDED COMMUNICATIONS
1 2 3	-	ng. It simplifies some coding, v atomic updates of local varia	and is very useful with accumulate operations, to ables. ( <i>End of rationale.</i> )
4 5 6 7 8	is the sam operation	e as for MPI_PROC_NULL in N	nk in all MPI RMA communication calls. The effect IPI point-to-point communication. After any RMA is still necessary to finish the RMA epoch with the e epoch.
9	11.3.1 P	ut	
10 11 12 13 14	and a mat		lar to the execution of a send by the origin process occess. The obvious difference is that all arguments buted by the origin process.
15 16 17	MPI_PUT(	origin_addr, origin_count, orig target_datatype, win)	in_datatype, target_rank, target_disp, target_count,
18	IN	origin_addr	initial address of origin buffer (choice)
19 20	IN	origin_count	number of entries in origin buffer (non-negative integer)
21	IN	origin_datatype	datatype of each entry in origin buffer (handle)
22 23	IN	target_rank	rank of target (non-negative integer)
24 25	IN	target_disp	displacement from start of window to target buffer (non-negative integer)
26 27	IN	target_count	number of entries in target buffer (non-negative integer)
28 29	IN	target_datatype	datatype of each entry in target buffer (handle)
30	IN	win	window object used for communication (handle)
31 32 33 34 35 26	int MPI_F	MPI_Aint target_disp	datatype, int target_rank,
36 37 38 39 40 41 42 43 44	TYPE( INTEC TYPE( INTEC TYPE(	<pre>target_disp, target_ (*), DIMENSION(), INTENT ER, INTENT(IN) :: origin (MPI_Datatype), INTENT(IN)</pre>	
45 46 47 48	<type< td=""><td></td><td>, ORIGIN_DATATYPE, TARGET_RANK, COUNT, TARGET_DATATYPE, WIN, IERROR) ) TARGET_DISP</td></type<>		, ORIGIN_DATATYPE, TARGET_RANK, COUNT, TARGET_DATATYPE, WIN, IERROR) ) TARGET_DISP

### INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR

Transfers origin_count successive entries of the type specified by the origin_datatype, starting at address origin_addr on the origin node, to the target node specified by the win, target_rank pair. The data are written in the target buffer at address target_addr = window_base+target_disp × disp_unit, where window_base and disp_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target_count and target_datatype. The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.

The communication must satisfy the same constraints as for a similar message-passing communication. The target_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window or in attached memory in a dynamic window.

The target_datatype argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the target datatype object was defined at the target process by the same sequence of calls used to define it at the origin process. The target datatype must contain only relative displacements, not absolute addresses. The same holds for get and accumulate operations.

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).

The performance of a put transfer can be significantly affected, on some systems, by the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI_ALLOC_MEM or MPI_WIN_ALLOCATE may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28

29

30

 31 

32

33

34

35

36 37

38

39

40

41

42

43

44

45

# $46 \\ 47$

CHAPTER 11. ONE-SIDED COMMUNICATIONS

12	11.3.2 G	et	
3			
4 5	MPI_GET(	origin_addr, origin_count, origin target_datatype, win)	n_datatype, target_rank, target_disp, target_count,
6 7	OUT	origin_addr	initial address of origin buffer (choice)
8 9	IN	origin_count	number of entries in origin buffer (non-negative integer)
10	IN	origin_datatype	datatype of each entry in origin buffer (handle)
11 12	IN	target_rank	rank of target (non-negative integer)
13 14	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)
15 16 17	IN	target_count	number of entries in target buffer (non-negative integer)
18	IN	target_datatype	datatype of each entry in target buffer (handle)
19	IN	win	window object used for communication (handle)
21 22 23 24 25 26 27 28 29 30 31 32 33 33 34	MPI_Get(c TYPE( INTEC TYPE( INTEC TYPE( INTEC	<pre>MPI_Aint target_disp MPI_Datatype target_d prigin_addr, origin_count, target_disp, target_d (*), DIMENSION(), ASYNCH GER, INTENT(IN) :: origin (MPI_Datatype), INTENT(IN) GER(KIND=MPI_ADDRESS_KIND) (MPI_Win), INTENT(IN) :: GER, OPTIONAL, INTENT(OUT)</pre>	<pre>datatype, int target_rank, , int target_count, datatype, MPI_Win win) , origin_datatype, target_rank, count, target_datatype, win, ierror) HRONOUS :: origin_addr n_count, target_rank, target_count 0 :: origin_datatype, target_datatype 0, INTENT(IN) :: target_disp win</pre>
<ol> <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> </ol>	<type INTEC INTEC Simila are copied overlappin target win</type 	TARGET_DISP, TARGET_G > ORIGIN_ADDR(*) GER(KIND=MPI_ADDRESS_KIND) GER ORIGIN_COUNT, ORIGIN_E TARGET_DATATYPE, WIN ar to MPI_PUT, except that from the target memory to g entries in the origin buffer.	COUNT, TARGET_DATATYPE, WIN, IERROR) TARGET_DISP DATATYPE, TARGET_RANK, TARGET_COUNT, N, IERROR the direction of data transfer is reversed. Data the origin. The origin_datatype may not specify The target buffer must be contained within the ry in a dynamic window, and the copied data must

# 11.3.3 Examples for Communication Calls

These examples show the use of the MPI_GET function. As all MPI RMA communication functions are nonblocking, they must be completed. In the following, this is accomplished with the routine MPI_WIN_FENCE, introduced in Section 11.5.

**Example 11.1** We show how to implement the generic indirect assignment A = B(map), where A, B, and map have the same distribution, and map is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER otype(p), oindex(m),
 & ! used to construct origin datatypes
 ttype(p), tindex(m),
 & ! used to construct target datatypes
 count(p), total(p),
 &
 disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
! This part does the work that depends on the locations of B.
! Can be reused while this does not change
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
 &
 comm, win, ierr)
! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change
! Compute number of entries to be received from each process
DO i=1,p
 count(i) = 0
END DO
DO i=1,m
 j = map(i)/m+1
 count(j) = count(j)+1
END DO
total(1) = 0
DO i=2,p
 total(i) = total(i-1) + count(i-1)
END DO
```

1 2

3

4

5 6

7

8

9 10

11

12 13

 $14 \\ 15$ 

16

17

18

19

20 21

22 23 24

25

26

27

28

29 30

31

32

33 34

35 36

37

38

39

40

41

42

43 44

45

46

47

```
1
 DO i=1,p
\mathbf{2}
 count(i) = 0
3
 END DO
4
\mathbf{5}
 ! compute origin and target indices of entries.
6
 ! entry i at current process is received from location
7
 ! k at process (j-1), where map(i) = (j-1)*m + (k-1),
8
 ! j = 1...p and k = 1...m
9
10
 DO i=1,m
^{11}
 j = map(i)/m+1
12
 k = MOD(map(i), m) + 1
13
 count(j) = count(j)+1
14
 oindex(total(j) + count(j)) = i
15
 tindex(total(j) + count(j)) = k
16
 END DO
17
18
 ! create origin and target datatypes for each get operation
19
 DO i=1,p
20
 CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
21
 oindex(total(i)+1:total(i)+count(i)), &
22
 MPI_REAL, otype(i), ierr)
23
 CALL MPI_TYPE_COMMIT(otype(i), ierr)
^{24}
 CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
25
 tindex(total(i)+1:total(i)+count(i)), &
26
 MPI_REAL, ttype(i), ierr)
27
 CALL MPI_TYPE_COMMIT(ttype(i), ierr)
28
 END DO
29
30
 ! this part does the assignment itself
^{31}
 CALL MPI_WIN_FENCE(0, win, ierr)
32
 disp_aint = 0
33
 DO i=1,p
34
 CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
35
 END DO
36
 CALL MPI_WIN_FENCE(0, win, ierr)
37
38
 CALL MPI_WIN_FREE(win, ierr)
39
 DO i=1,p
40
 CALL MPI_TYPE_FREE(otype(i), ierr)
41
 CALL MPI_TYPE_FREE(ttype(i), ierr)
42
 END DO
43
 RETURN
44
 END
45
46
 Example 11.2
47
48
```

A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
 &
 comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
 j = map(i)/m
 disp_aint = MOD(map(i),m)
 CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

### 11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather than replacing it. This will allow, for example, the accumulation of a sum by having all involved processes add their contributions to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 11.7 for details.

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10 11

12

13

14

15

16 17

18

19

20

21

22

23

 24 

25

26

27 28

29 30

 31 

32

33

34

35

1Accumulate Function  $\mathbf{2}$ 3 4 MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, 5target_count, target_datatype, op, win) 6 IN origin_addr initial address of buffer (choice) 7 IN origin_count 8 number of entries in buffer (non-negative integer) 9 IN origin_datatype datatype of each entry (handle) 10 IN target_rank rank of target (non-negative integer) 11 IN target_disp displacement from start of window to beginning of tar-12get buffer (non-negative integer) 13 14IN target_count number of entries in target buffer (non-negative inte-15ger) 16IN target_datatype datatype of each entry in target buffer (handle) 17IN reduce operation (handle) 18 op 19IN win window object (handle) 2021int MPI_Accumulate(const void *origin_addr, int origin_count, 22 MPI_Datatype origin_datatype, int target_rank, 23MPI_Aint target_disp, int target_count,  24 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win) 2526MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, ierror) 27TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 28INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 29 30 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 3132 TYPE(MPI_Op), INTENT(IN) :: op 33 TYPE(MPI_Win), INTENT(IN) :: win 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 36 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR) 37 <type> ORIGIN_ADDR(*) 38 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 39 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 40 TARGET_DATATYPE, OP, WIN, IERROR 41 42Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count, and 43origin_datatype) to the buffer specified by arguments target_count and target_datatype, at 44offset target_disp, in the target window specified by target_rank and win, using the operation 45op. This is like MPI_PUT except that data is combined into the target area instead of 46overwriting it. 47Any of the predefined operations for MPI_REDUCE can be used. User-defined functions 48

cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added

to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both datatype arguments must be constructed from the same predefined datatype. The operation **op** applies to elements of that predefined type. The parameter **target_datatype** must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative function f(a, b) = b; i.e., the current value in the target memory is replaced by the value supplied by the origin.

MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE, MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not in collective reduction operations such as MPI_REDUCE.

Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

**Example 11.3** We want to compute  $B(j) = \sum_{map(i)=j} A(i)$ . The arrays A, B, and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr, disp_int
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
size = m * realextent
disp_int = realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
 comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
 j = map(i)/m
 disp_aint = MOD(map(i),m)
 CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
 &
 MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 11.2, except that a call to get has been ⁴⁶ replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes  $B = A(map^{-1})$ , which is the reverse assignment to the one computed in that previous 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

36

37

38

39

40

41 42

43

44

example.) In a similar manner, we can replace in Example 11.1, the call to get by a call to
 accumulate, thus performing the computation with only one communication between any
 two processes.

4 5

6

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.

12

```
 <sup>13</sup> 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		0 51 1	,
17	IN	origin_addr	initial address of buffer (choice)
18 19	IN	origin_count	number of entries in origin buffer (non-negative integer)
20 21	IN	origin_datatype	datatype of each entry in origin buffer (handle)
22	OUT	result_addr	initial address of result buffer (choice)
23 24	IN	result_count	number of entries in result buffer (non-negative integer)
25 26	IN	result_datatype	datatype of each entry in result buffer (handle)
20	IN	target_rank	rank of target (non-negative integer)
28 29	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)
30 31 32	IN	target_count	number of entries in target buffer (non-negative integer)
33	IN	target_datatype	datatype of each entry in target buffer (handle)
34	IN	ор	reduce operation (handle)
35 36	IN	win	window object (handle)
37 38 39 40 41 42	int MPI_	MPI_Datatype origin int result_count, MI int target_rank, MPI	d *origin_addr, int origin_count, _datatype, void *result_addr, PI_Datatype result_datatype, I_Aint target_disp, int target_count, _datatype, MPI_Op op, MPI_Win win)
43 44 45 46 47 48	TYPE TYPE	<pre>result_count, result target_count, target (*), DIMENSION(), INTEN (*), DIMENSION(), ASYNC</pre>	<pre>prigin_count, origin_datatype, result_addr, t_datatype, target_rank, target_disp, t_datatype, op, win, ierror) NT(IN), ASYNCHRONOUS :: origin_addr CHRONOUS :: result_addr In_count, result_count, target_rank,</pre>

target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
result_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
<pre>MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,</pre>

Accumulate origin_count elements of type origin_datatype from the origin buffer ( origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation, specified by target_disp, target_count, and target_datatype. The data transferred from origin to target must fit, without truncation, in the target buffer. Likewise, the data copied from target to origin must fit, without truncation, in the result buffer.

The origin and result buffers (origin_addr and result_addr) must be disjoint. Each datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 11.7 for details.

Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE can be specified as op. User-defined functions cannot be used. A new predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function f(a, b) = a; i.e., the current value in the target memory is returned in the result buffer at the origin and no operation is performed on the target buffer. When MPI_NO_OP is specified as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored. MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others.

Advice to users. MPI_GET is similar to MPI_GET_ACCUMULATE, with the operation MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

### Fetch and Op Function

The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetchand-increment or fetch-and-add calls that might be supported by special hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used

### Unofficial Draft for Comment Only

 24 

 31 

```
454
 CHAPTER 11. ONE-SIDED COMMUNICATIONS
1
 subset of the functionality of MPI_GET_ACCUMULATE.
\mathbf{2}
3
 MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win)
4
5
6
 IN
 origin_addr
 initial address of buffer (choice)
7
 OUT
 result_addr
 initial address of result buffer (choice)
8
 IN
 datatype
 datatype of the entry in origin, result, and target buf-
9
 fers (handle)
10
11
 IN
 target_rank
 rank of target (non-negative integer)
12
 IN
 target_disp
 displacement from start of window to beginning of tar-
13
 get buffer (non-negative integer)
14
 IN
 op
 reduce operation (handle)
15
16
 IN
 window object (handle)
 win
17
18
 int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
19
 MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
20
 MPI_Op op, MPI_Win win)
21
 MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
22
 target_disp, op, win, ierror)
23
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
24
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
25
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
 INTEGER, INTENT(IN) :: target_rank
27
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
28
 TYPE(MPI_Op), INTENT(IN) :: op
29
 TYPE(MPI_Win), INTENT(IN) :: win
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
33
 TARGET_DISP, OP, WIN, IERROR)
34
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
35
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
36
 INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
37
 Accumulate one element of type datatype from the origin buffer (origin_addr) to the
38
 buffer at offset target_disp, in the target window specified by target_rank and win, using
39
 the operation op and return in the result buffer result_addr the content of the target buffer
40
 before the accumulation.
41
 The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the
42
 predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be
43
 specified as op; user-defined functions cannot be used. The datatype argument must be a
44
 predefined datatype. The operation is executed atomically.
45
46
47
48
```

Compare	and Swap Function	1
compare	_	tomic compare and swap where the value at the origin is get, which is atomically replaced by a third value only if re equal.
MPI_CO	MPARE_AND_SWAP(ori target_disp, win)	gin_addr, compare_addr, result_addr, datatype, target_rank,
IN	origin_addr	initial address of buffer (choice)
IN	compare_addr	initial address of compare buffer (choice) 1
OUT	result_addr	initial address of result buffer (choice)
IN	datatype	datatype of the element in all buffers (handle) $1$
IN	target_rank	rank of target (non-negative integer) ¹
IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)
IN	win	window object (handle)
TYP TYP TYP INT INT TYP	<pre>target_rank, t E(*), DIMENSION(), E(*), DIMENSION(), E(*), DIMENSION(), E(MPI_Datatype), INTE EGER, INTENT(IN) ::</pre>	target_rank S_KIND), INTENT(IN) :: target_disp
<ty INT INT This compare target_ra origin_ac the targ</ty 	TARGET_RANK, T. pe> ORIGIN_ADDR(*), ( EGER(KIND=MPI_ADDRESS EGER DATATYPE, TARGET s function compares one _addr with the buffer at nk and win and replaces dr if the compare buffer et is returned in the buf	ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE, 3 ARGET_DISP, WIN, IERROR) 3 COMPARE_ADDR(*), RESULT_ADDR(*) 3 S_KIND) TARGET_DISP 3

Multi-language types, or Byte as specified in Section 5.9.2. The origin and result buffers

(origin_addr and result_addr) must be disjoint.

46

47

1	11 2 5 1	Dogwoot based DMA Comm	unication Operations		
2					
3	-	Request-based RMA communication operations allow the user to associate a request handle			
4		with the RMA operations and test or wait for the completion of these requests using the			
5		functions described in Section 3.7.3. Request-based RMA operations are only valid within a passive target epoch (see Section 11.5).			
6			ion call in which an RMA operation completes, the		
7	-		atus object is set appropriately (see Section 3.2.5). All		
8 9			s of status query functions (e.g., MPI_GET_COUNT)		
10	are under	fined. It is valid to mix di	fferent request types (e.g., any combination of RMA		
11		, -	lests, generalized requests, or point-to-point requests)		
12		—	npletions (e.g., MPI_WAITALL). It is erroneous to call		
13		QUESI_FREE or MPI_CANC uests are not persistent.	CEL for a request associated with an RMA operation.		
14	-	-	icit bulk synchronization using		
15			I_ALL, MPI_WIN_FLUSH_LOCAL, or		
16 17		·	indicates completion of the RMA operations. How-		
18			he request handle to allow the MPI implementation to		
19	-	-	h these requests; in such cases the wait operation will		
20	complete	locally.			
21					
22	MPI_RPU	JT(origin_addr, origin_count,	origin_datatype, target_rank, target_disp, target_count,		
23 24		target_datatype, win,	request)		
25	IN	origin_addr	initial address of origin buffer (choice)		
26 27	IN	origin_count	number of entries in origin buffer (non-negative integer)		
28 29	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
30	IN	target_rank	rank of target (non-negative integer)		
31 32	IN	target_disp	displacement from start of window to target buffer (non-negative integer)		
33 34	IN	target_count	number of entries in target buffer (non-negative integer)		
35 36	IN	target_datatype	datatype of each entry in target buffer (handle)		
37	IN	win	window object used for communication (handle)		
38	OUT	request	RMA request (handle)		
39					
40 41	int MPI_	_Rput(const void *origin	_addr, int origin_count,		
42			in_datatype, int target_rank,		
43		-	isp, int target_count,		
44	MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)				
45					
46	MPI_Rput	MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,			
47 48		<pre>target_disp, target_count, target_datatype, win, request, ierror)</pre>			
		TETTOT			

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
 1
 \mathbf{2}
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
 3
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
 4
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 TYPE(MPI_Win), INTENT(IN) :: win
 5
 6
 TYPE(MPI_Request), INTENT(OUT) :: request
 7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 8
MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
 9
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
 10
 IERROR)
 11
 <type> ORIGIN_ADDR(*)
 12
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 13
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
 14
 TARGET_DATATYPE, WIN, REQUEST, IERROR
 15
 16
```

MPI_RPUT is similar to MPI_PUT (Section 11.3.1), except that it allocates a communication request object and associates it with the request handle (the argument request). The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) indicates that the sender is now free to update the locations in the origin buffer. It does not indicate that the data is available at the target window. If remote completion is required, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL can be used.

MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

OUT	origin_addr	initial address of origin buffer (choice)	27
IN	origin_count	number of entries in origin buffer (non-negative inte-	28 29
		ger)	30
IN	origin_datatype	datatype of each entry in origin buffer (handle)	31
IN	target_rank	rank of target (non-negative integer)	32
IN	target_disp	displacement from window start to the beginning of	33 34
		the target buffer (non-negative integer)	35
IN	target_count	number of entries in target buffer (non-negative inte- ger)	36 37
IN	target_datatype	datatype of each entry in target buffer (handle)	38
IN	win	window object used for communication (handle)	39
OUT	request	RMA request (handle)	40 41
		/	42

int MPI_Rget(void *origin_addr, int origin_count,

```
MPI_Datatype origin_datatype, int target_rank,44MPI_Aint target_disp, int target_count,45MPI_Datatype target_datatype, MPI_Win win,46MPI_Request *request)47
```

Unofficial Draft for Comment Only

17

18

19

20

21

22

23 24

25

26

1	MPI_Rget	t(origin_addr, origin	_count, origin_datatype, target_rank,		
2			rget_count, target_datatype, win, request,		
3		ierror)			
4 5			ASYNCHRONOUS :: origin_addr		
6			origin_count, target_rank, target_count NT(IN) :: origin_datatype, target_datatype		
7			_KIND), INTENT(IN) :: target_disp		
8		E(MPI_Win), INTENT(IN	<b>o i</b>		
9		TYPE(MPI_Request), INTENT(OUT) :: request			
10	INTE	EGER, OPTIONAL, INTEN	T(OUT) :: ierror		
11	MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,				
12	III 1_10001		RGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,		
13 14		IERROR)			
14 15	<typ< td=""><td><pre>De&gt; ORIGIN_ADDR(*)</pre></td><td></td></typ<>	<pre>De&gt; ORIGIN_ADDR(*)</pre>			
16	INTE	EGER(KIND=MPI_ADDRESS	_KIND) TARGET_DISP		
17	INTE	-	IGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,		
18		TARGET_DATATYP	E, WIN, REQUEST, IERROR		
19	MPI_RGET is similar to MPI_GET (Section 11.3.2), except that it allocates a commu-				
20			eiates it with the request handle (the argument request)		
21			r completion. The completion of an MPI_RGET operation		
22 23			le in the origin buffer. If origin_addr points to memory		
23	attached	to a window, then the da	ata becomes available in the private copy of this window.		
25					
26	MPI_RAG	CCUMULATE(origin_addr	, origin_count, origin_datatype, target_rank, target_disp,		
27		target_count, target_	et_datatype, op, win, request)		
28 29	IN	origin_addr	initial address of buffer (choice)		
30	IN	origin_count	number of entries in buffer (non-negative integer)		
31	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
32	IN	target_rank	rank of target (non-negative integer)		
33 34	IN	target_disp	displacement from start of window to beginning of tar-		
35			get buffer (non-negative integer)		
36	IN	target_count	number of entries in target buffer (non-negative inte-		
37			$\operatorname{ger})$		
38	IN	target_datatype	datatype of each entry in target buffer (handle)		
39 40	IN	ор	reduce operation (handle)		
41	IN	win	window object (handle)		
42	OUT	request	RMA request (handle)		
43					
44	int MPI_	_Raccumulate(const vo	id *origin_addr, int origin_count,		
45		MPI_Datatype origin_datatype, int target_rank,			
46 47	MPI_Aint target_disp, int target_count,				
47		MPI_Datatype ta MPI_Request *re	rget_datatype, MPI_Op op, MPI_Win win,		
		mri_request *re			

MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,	1	
target_disp, target_count, target_datatype, op, win, request,	2	
ierror)	3	
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr	4	
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	5	
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	6	
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	7	
TYPE(MPI_Op), INTENT(IN) :: op	8	
TYPE(MPI_Win), INTENT(IN) :: win	9	
TYPE(MPI_Request), INTENT(OUT) :: request	10	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11	
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	12 13	
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR)	14	
<type> ORIGIN_ADDR(*)</type>	15	
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	16	
INTEGER (KIND-MFI_ADDRESS_KIND) TARGET_DISP INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	17	
TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	18	
IARGEI_DAIAIIPE, UP, WIN, REQUESI, IERRUR	19	
MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that	20	
it allocates a communication request object and associates it with the request handle (the	21	

MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.

1MPI_RGET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,  $\mathbf{2}$ result_count, result_datatype, target_rank, target_disp, target_count, 3 target_datatype, op, win, request) 4 IN origin_addr initial address of buffer (choice) 5IN origin_count number of entries in origin buffer (non-negative inte-6 ger) 7 8 IN origin_datatype datatype of each entry in origin buffer (handle) 9 OUT result addr initial address of result buffer (choice) 10 IN result_count number of entries in result buffer (non-negative inte-11 ger) 1213IN result_datatype datatype of each entry in result buffer (handle) 14IN target_rank rank of target (non-negative integer) 15IN target_disp displacement from start of window to beginning of tar-16 get buffer (non-negative integer) 1718 IN target_count number of entries in target buffer (non-negative inte-19 ger) 20IN target_datatype datatype of each entry in target buffer (handle) 21IN ор reduce operation (handle) 22 23IN win window object (handle)  24 OUT RMA request (handle) request 2526int MPI_Rget_accumulate(const void *origin_addr, int origin_count, 27MPI_Datatype origin_datatype, void *result_addr, 28int result_count, MPI_Datatype result_datatype, 29 int target_rank, MPI_Aint target_disp, int target_count, 30 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,  31 MPI_Request *request) 32 33MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype, 34 result_addr, result_count, result_datatype, target_rank, 35 target_disp, target_count, target_datatype, op, win, request, 36 ierror) 37 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr 38 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr 39 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank, 40 target_count 41 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype, 42result_datatype 43 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 44 TYPE(MPI_Op), INTENT(IN) :: op 45TYPE(MPI_Win), INTENT(IN) :: win 46 TYPE(MPI_Request), INTENT(OUT) :: request 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
 IERROR)
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
 IERROR
```

MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

#### Memory Model 11.4

The memory semantics of RMA are best understood by using the concept of *public* and 20private window copies. We assume that systems have a public memory region that is 21addressable by all processes (e.g., the shared memory in shared memory machines or the 22 exposed main memory in distributed memory machines). In addition, most machines have 23fast private buffers (e.g., transparent caches or explicit communication buffers) local to each process where copies of data elements from the main memory can be stored for faster 25access. Such buffers are either coherent, i.e., all updates to main memory are reflected in 26all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory 27need to be synchronized and updated in all private copies explicitly. Coherent systems 28allow direct updates to remote memory without any participation of the remote side. Non-29coherent systems, however, need to call RMA functions in order to reflect updates to the 30 public window in their private memory. Thus, in coherent memory, the public and the  31 private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two **memory models** called **RMA unified**, if public and 33 private window are logically identical, and **RMA separate**, otherwise. 34

In the RMA separate model, there is only one instance of each variable in process 35memory, but a distinct *public* copy of the variable for each window that contains it. A load 36 accesses the instance in process memory (this includes MPI sends). A local store accesses 37 and updates the instance in process memory (this includes MPI receives), but the update 38 may affect other public copies of the same locations. A get on a window accesses the public 39 copy of that window. A put or accumulate on a window accesses and updates the public 40 copy of that window, but the update may affect the private copy of the same locations 41 in process memory, and public copies of other overlapping windows. This is illustrated in 42Figure 11.1. 43

In the RMA unified model, public and private copies are identical and updates via put 44or accumulate calls are eventually observed by load operations without additional RMA 45calls. A store access to a window is eventually visible to remote get or accumulate calls 46without additional RMA calls. These stronger semantics of the RMA unified model allow 47the user to omit some synchronization calls and potentially improve performance. 48

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

1 2

3

4

5

6  $\overline{7}$ 

8 9

10

11

12

13

14

15

1617

18

19

 24 

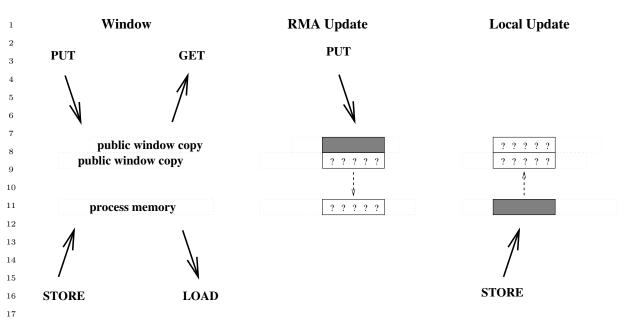


Figure 11.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows.

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.*)

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.

# Unofficial Draft for Comment Only

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

These calls are used for active target communication. An access epoch is started ⁴² at the origin process by a call to MPI_WIN_START and is terminated by a call to ⁴³ MPI_WIN_COMPLETE. The start call has a group argument that specifies the group ⁴⁴ of target processes for that epoch. An exposure epoch is started at the target process ⁴⁵ by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post ⁴⁶ call has a group argument that specifies the set of origin processes for that epoch. ⁴⁷

### 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 30

 31 

32 33

34

35

36

37

38

39

40

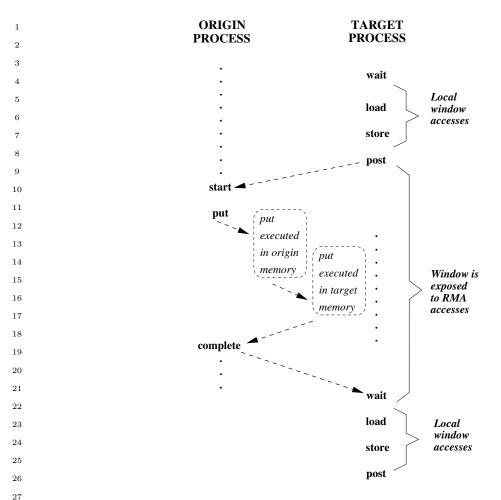


Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

29 30 31

32

33 34

35

36 37

38

39

40

28

3. Finally, shared lock access is provided by the functions MPI_WIN_LOCK,

MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL.

MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide exclusive lock capability. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

These four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively.

Figure 11.2 illustrates the general synchronization pattern for active target communi-41 cation. The synchronization between **post** and **start** ensures that the put call of the origin 42process does not start until the target process exposes the window (with the **post** call); 43 the target process will expose the window only after preceding local accesses to the window 44have completed. The synchronization between complete and wait ensures that the put call 45of the origin process completes before the window is unexposed (with the wait call). The 4647target process will execute following local accesses to the target window only after the wait returned. 48

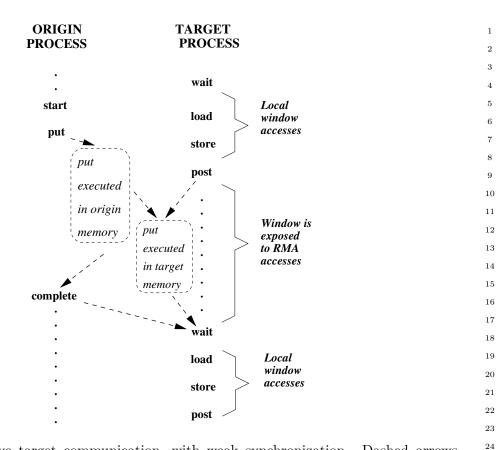


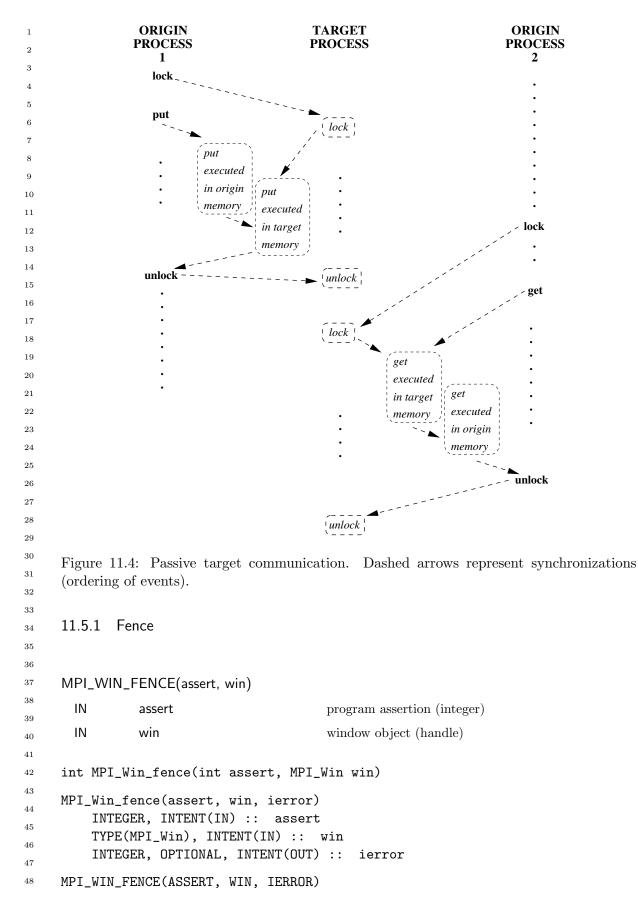
Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

Figure 11.2 shows operations occurring in the natural temporal order implied by the synchronizations: the post occurs before the matching start, and complete occurs before the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as illustrated in Figure 11.3. The access to the target window is delayed until the window is exposed, after the post. However the start may complete earlier; the put and complete may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

*Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End* of rationale.)

### Unofficial Draft for Comment Only



### INTEGER ASSERT, WIN, IERROR

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and the local process issued RMA communication calls on win between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait.

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a call with assert equal to MPI_MODE_NOPRECEDE) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert = 0 is always valid.

Advice to users. Calls to MPI_WIN_FENCE should both precede and follow calls to RMA communication functions that are synchronized with fence calls. (*End of advice to users.*)

### 11.5.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)	33
IN	assert	program assertion (integer)	34
IN	win	window object (handle)	35
			36
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)			37
IIIC PI	Int MFI_WIN_Start(MFI_Group group, Int assert, MFI_WIN WIN)		
MPI_W	MPI_Win_start(group, assert, win, ierror)		
TYPE(MPI_Group), INTENT(IN) :: group			40
INTEGER, INTENT(IN) :: assert			41
T	TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
I			
			44
_	MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR		
1.			

Starts an RMA access epoch for win. RMA calls issued on win during this epoch must 47 access only windows at processes in group. Each process in group must issue a matching 48

### Unofficial Draft for Comment Only

1call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary,  $\mathbf{2}$ until the target process executed the matching call to MPI_WIN_POST. MPI_WIN_START 3 is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not 4 required to. 5The assert argument is used to provide assertions on the context of the call that may 6 be used for various optimizations. This is described in Section 11.5.5. A value of assert =  $\overline{7}$ 0 is always valid. 8 9 MPI_WIN_COMPLETE(win) 10 11IN window object (handle) win 1213int MPI_Win_complete(MPI_Win win) 14MPI_Win_complete(win, ierror) 15TYPE(MPI_Win), INTENT(IN) :: win 16INTEGER, OPTIONAL, INTENT(OUT) :: 17ierror 18 MPI_WIN_COMPLETE(WIN, IERROR) 19INTEGER WIN, IERROR 2021Completes 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 22 when the call returns. 23MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin, but  24 not at the target. A put or accumulate call may not have completed at the target when it 2526has completed at the origin. Consider the sequence of calls in the example below. 2728Example 11.4 29 30 MPI_Win_start(group, flag, win);  31 MPI_Put(..., win); 32 MPI_Win_complete(win); 33 34The call to MPI_WIN_COMPLETE does not return until the put call has completed 35 at the origin; and the target window will be accessed by the put operation only after the 36 call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process. 37 This still leaves much choice to implementors. The call to MPI_WIN_START can block 38 until the matching call to MPI_WIN_POST occurs at all target processes. One can also 39 have implementations where the call to MPI_WIN_START is nonblocking, but the call to 40MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations 41 where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks 42until the call to MPI_WIN_POST occurred; or even implementations where all three calls 43can complete before any target process has called MPI_WIN_POST — the data put must 44be buffered, in this last case, so as to allow the put to complete at the origin ahead of its

⁴⁵ completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence
 ⁴⁶ above must complete, without further dependencies.

MPI_WIN_POST(group, assert, win)			1
IN	group	group of origin processes (handle)	2
IN	assert	program assertion (integer)	3 4
			4 5
IN	win	window object (handle)	6
			7
int MF	PI_Win_post(MPI_Gr	coup group, int assert, MPI_Win win)	8
MPI_Win_post(group, assert, win, ierror)			9
ΤY	PE(MPI_Group), IN	NTENT(IN) :: group	10
IN	NTEGER, INTENT(IN)	:: assert	11
ΤY	PE(MPI_Win), INTE	ENT(IN) :: win	12
IN	NTEGER, OPTIONAL,	INTENT(OUT) :: ierror	13
MPT WI	N POST (GROUP ASS	SERT, WIN, IERROR)	14
	ITEGER GROUP, ASSE		15
	-		16
		e epoch for the local window associated with win. Only processes	17
-		window with RMA calls on win during this epoch. Each process	18
ın grou	p must issue a match	ning call to MPI_WIN_START. MPI_WIN_POST does not block.	19
			20
MPI_W	/IN_WAIT(win)		21
IN	win	window object (handle)	22 23
IIN	VVIII	window object (nandle)	23
int MT			25
int MF	PI_Win_wait(MPI_Wi		26
MPI_Wi	n_wait(win, ierro	or)	27
ΤY	PE(MPI_Win), INTE	ENT(IN) :: win	28
IN	ITEGER, OPTIONAL,	INTENT(OUT) :: ierror	29
мрт ш	N_WAIT(WIN, IERRO	IR )	30
	TEGER WIN, IERROF		31
			32
		posure epoch started by a call to MPI_WIN_POST on win. This	33
		/IN_COMPLETE(win) issued by each of the origin processes that	34
were or	ranted access to the w	undow during this enoch. The call to MPL WIN WALL will block	

call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 11.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

 $45 \\ 46$ 

35

36

37

38

39

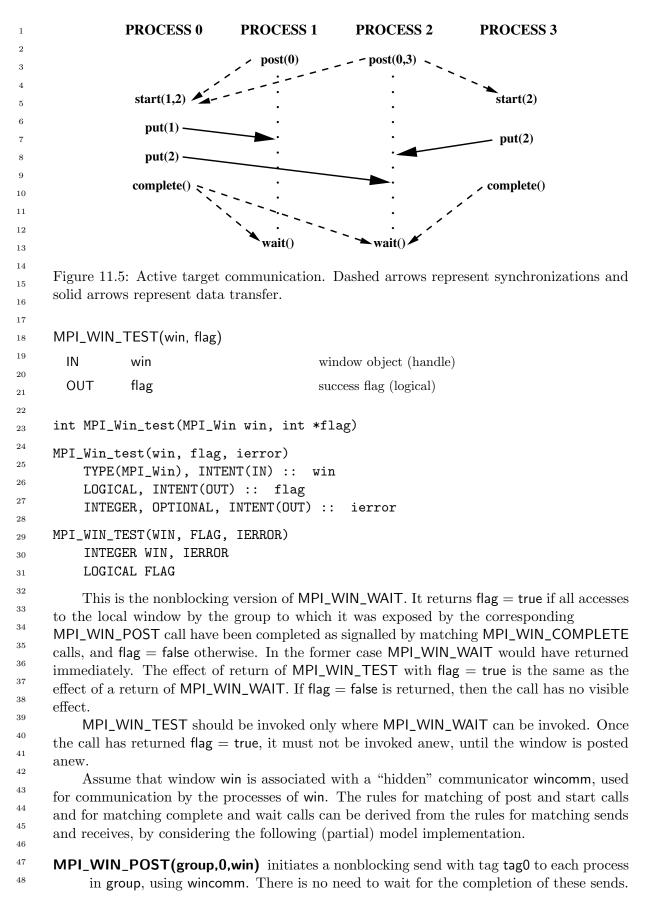
40

41

42

43

- 47
- 48



- MPI_WIN_START(group,0,win) initiates 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)** initiates 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)** initiates a nonblocking receive with tag **tag1** from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice versa.

*Rationale.* The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph  $G = \langle V, E \rangle$ , where  $V = \{0, \ldots, n-1\}$  and  $ij \in E$  if origin process *i* accesses the window at target process *j*. Then each process *i* issues a call to MPI_WIN_POST(*ingroup*_i, ...), followed by a call to MPI_WIN_START(*outgroup*_i,...), where *outgroup*_i =  $\{j : ij \in E\}$  and *ingroup*_i =

 $\{j : ji \in E\}$ . A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

# 11.5.3 Lock

MPI_WIN_LOCK(lock_type, rank, assert, win)

IN	lock_type	either $MPI_LOCK_EXCLUSIVE$ or	41
		MPI_LOCK_SHARED (state)	42
IN	rank	rank of locked window (non-negative integer)	43
IN	assert	program assertion (integer)	44
	assert	program assertion (integer)	45
IN	win	window object (handle)	46
			47

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

#### Unofficial Draft for Comment Only

 31 

```
1
 MPI_Win_lock(lock_type, rank, assert, win, ierror)
\mathbf{2}
 INTEGER, INTENT(IN) :: lock_type, rank, assert
3
 TYPE(MPI_Win), INTENT(IN) :: win
4
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
5
 MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
6
 INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
7
8
 Starts an RMA access epoch. The window at the process with rank rank can be accessed
9
 by RMA operations on win during that epoch. Multiple RMA access epochs (with calls
10
 to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a
11
 different process.
12
13
 MPI_WIN_LOCK_ALL(assert, win)
14
15
 IN
 program assertion (integer)
 assert
16
 IN
 win
 window object (handle)
17
18
 int MPI_Win_lock_all(int assert, MPI_Win win)
19
20
 MPI_Win_lock_all(assert, win, ierror)
21
 INTEGER, INTENT(IN) :: assert
22
 TYPE(MPI_Win), INTENT(IN) :: win
23
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
^{24}
 MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
25
 INTEGER ASSERT, WIN, IERROR
26
27
 Starts an RMA access epoch to all processes in win, with a lock type of
28
 MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on
29
 all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL
30
 must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL
^{31}
 refers to a lock on all members of the group of the window.
32
33
 Advice to users.
 There may be additional overheads associated with using
34
 MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These
35
 overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when
36
 possible (see Section 11.5.5). (End of advice to users.)
37
38
39
 MPI_WIN_UNLOCK(rank, win)
40
41
 IN
 rank of window (non-negative integer)
 rank
42
 IN
 window object (handle)
 win
43
44
 int MPI_Win_unlock(int rank, MPI_Win win)
45
46
 MPI_Win_unlock(rank, win, ierror)
47
 INTEGER, INTENT(IN) :: rank
48
 TYPE(MPI_Win), INTENT(IN) ::
 win
```

#### Unofficial Draft for Comment Only

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
	2
MPI_WIN_UNLOCK(RANK, WIN, IERROR)	3
INTEGER RANK, WIN, IERROR	4
Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window win.	5
RMA operations issued during this period will have completed both at the origin and at the	6
target when the call returns.	7
	8
	9
MPI_WIN_UNLOCK_ALL(win)	10
IN win window object (handle)	11
	12
int MPI_Win_unlock_all(MPI_Win win)	13
	14
MPI_Win_unlock_all(win, ierror)	15
TYPE(MPI_Win), INTENT(IN) :: win	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_WIN_UNLOCK_ALL(WIN, IERROR)	18
INTEGER WIN, IERROR	19
	20
Completes a shared RMA access epoch started by a call to $MPI_WIN_LOCK_ALL$ on	21
window win. RMA operations issued during this epoch will have completed both at the	22

window win. RMA operations issued during this epoch will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock calls, and to protect load/store accesses to a locked local or shared memory window executed between the lock and unlock calls. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. For example, a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

*Rationale.* An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (*End of advice to users.*)

47 48

23 24

25

26

27

28

29

30

 31 

32

33

34

35 36

37

38

39

40

41

42

43 44

45

46

#### Unofficial Draft for Comment Only

1 2 3 4 5	lock call MPI_WI	ls to windows in N_ALLOCATE	n memory allocate (Section $11.2.2$ ), o	of RMA communication that is synchronized by ed by MPI_ALLOC_MEM (Section 8.2), or attached with MPI_WIN_ATTACH bly only in such memory.
6 7 8 9 10 11	is im all na	not shared may plemented more located memory tural to impose	v require an asynce easily, and can ac It can be avoide	of passive target communication when memory chronous software agent. Such an agent can be hieve better performance, if restricted to specially ed altogether if shared memory is used. It seems allows one to use shared memory for third party machines.
12 13	(E	End of rationale.	)	
14 15	Cor	nsider the seque	nce of calls in the	example below.
16	Examp	le 11.5		
17 18 19 20	MPI_Put	n_lock(MPI_LOC ;(, rank, . n_unlock(rank,	, win);	ank, assert, win);
21 22 23 24 25 26 27 28 29	The call to MPI_WIN_UNLOCK will not return until the put transfer has completed at the origin and at the target. This still leaves much freedom to implementors. The call to MPI_WIN_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired — the 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.			
30 31	11.5.4	Flush and Syne		
32 33	All flush	n and sync funct	ions can be called	only within passive target epochs.
34 35	MPI_WI	N_FLUSH(rank,	win)	
36	IN	rank		rank of target window (non-negative integer)
37 38	IN	win		window object (handle)
39 40	int MPI	[_Win_flush(ir	t rank, MPI_Wir	n win)
41 42 43 44 45 46 47 48	INT TYF INT MPI_WIN	-	IN) :: rank NTENT(IN) :: w L, INTENT(OUT) WIN, IERROR)	

MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target.

MPI_WIN_FLUSH_ALL(win)			
IN	win	window object (handle)	7
			8
int MP	I_Win_flush_all(	(MPT Win win)	9
			10
	n_flush_all(win,		11
	PE(MPI_Win), INT		12
ΙN	IEGER, UPIIUNAL,	, INTENT(OUT) :: ierror	13
MPI_WI	N_FLUSH_ALL(WIN,	, IERROR)	14
IN	TEGER WIN, IERRO	IR	15 16
Al	RMA operations is	ssued by the calling process to any target on the specified window	10
	*	e specified window will have completed both at the origin and at	18
-	get when this call r	* * 0	19
(	<u></u>		20
			21
MPI_W	IN_FLUSH_LOCAL	.(rank, win)	22
IN	rank	rank of target window (non-negative integer)	23
IN	win	window object (handle)	24
			25
int MP	T Win flush loca	al(int rank, MPI_Win win)	26
			27
		ank, win, ierror)	28
	TEGER, INTENT(IN		29
	PE(MPI_Win), INT		30 31
ΤΝ	IEGER, UPIIUNAL,	, INTENT(OUT) :: ierror	32
MPI_WI	N_FLUSH_LOCAL(RA	ANK, WIN, IERROR)	33
IN	TEGER RANK, WIN,	IERROR	34
Lo	cally completes at t	the origin all outstanding RMA operations initiated by the calling	35
	° *	ess specified by rank on the specified window. For example, after	36
		e user may reuse any buffers provided to put, get, or accumulate	37
operati			38
			39
			40
WPI_W	'IN_FLUSH_LOCAL	ALL(WIN)	41
IN	win	window object (handle)	42
			43
int MP	I_Win_flush_loca	al_all(MPI_Win win)	$44 \\ 45$
MPJ Wi	n_flush_local_al	l(win. ierror)	45 46
		TENT(IN) :: win	47
	· _ · ·	, INTENT(OUT) :: ierror	48
	-		

**Unofficial Draft for Comment Only** 

1

 $\mathbf{2}$ 

```
1
 MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
\mathbf{2}
 INTEGER WIN, IERROR
3
 All RMA operations issued to any target prior to this call in this window will have
4
 completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.
5
6
\overline{7}
 MPI_WIN_SYNC(win)
8
 IN
 window object (handle)
 win
9
10
 int MPI_Win_sync(MPI_Win win)
11
12
 MPI_Win_sync(win, ierror)
13
 TYPE(MPI_Win), INTENT(IN) ::
 win
14
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
15
16
 MPI_WIN_SYNC(WIN, IERROR)
17
 INTEGER WIN, IERROR
18
 The call MPI_WIN_SYNC synchronizes the private and public window copies of win.
19
 For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the
20
 effect of ending and reopening an access and exposure epoch on the window (note that it
21
 does not actually end an epoch or complete any pending MPI RMA operations).
22
23
 11.5.5
 Assertions
^{24}
25
 The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE,
26
 MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL is used to provide assertions on the context of
27
 the call that may be used to optimize performance. The assert argument does not change
28
 program semantics if it provides correct information on the program — it is erroneous to
29
 provide incorrect information. Users may always provide assert = 0 to indicate a general
30
 case where no guarantees are made.
31
 Advice to users. Many implementations may not take advantage of the information
32
 in assert; some of the information is relevant only for noncoherent shared memory ma-
33
 chines. Users should consult their implementation's manual to find which information
34
 is useful on each system. On the other hand, applications that provide correct asser-
35
 tions whenever applicable are portable and will take advantage of assertion specific
36
 optimizations whenever available. (End of advice to users.)
37
38
 Implementations can always ignore the
 Advice to implementors.
39
 assert argument. Implementors should document which assert values are significant
40
 on their implementation. (End of advice to implementors.)
41
42
 assert is the bit-vector OR of zero or more of the following integer constants:
43
 MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT,
44
 MPI_MODE_NOPRECEDE, and MPI_MODE_NOSUCCEED. The significant options are listed
45
 below for each call.
46
47
 Advice to users. C/C++ users can use bit vector or () to combine these constants;
48
 Fortran 90 users can use the bit-vector IOR intrinsic. 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:

- ve not yet oc-
- 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.
- 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 conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

Advice to users. Note that the nostore and noprecede flags provide information on what happened *before* the call; the noput and nosucceed flags provide information on what will happen *after* the call. (*End of advice to users.*)

#### 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

# 11.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message-passing, datatypes must be committed before they can be used in RMA communication.

#### 8 9

1

 $\mathbf{2}$ 

3

4

5

6

7

10

11

17

18 19

20

24

48

# 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).

# 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.

24		
25	MPI_ERR_WIN	invalid win argument
26	MPI_ERR_BASE	invalid <b>base</b> argument
27	MPI_ERR_SIZE	invalid size argument
28	MPI_ERR_DISP	invalid disp argument
29	MPI_ERR_LOCKTYPE	invalid locktype argument
30	MPI_ERR_ASSERT	invalid assert argument
31	MPI_ERR_RMA_CONFLICT	conflicting accesses to window
32	MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls
33	MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case
34		of a window created with
35		MPI_WIN_CREATE_DYNAMIC, target memory is not
36		attached)
37	MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource
38		exhaustion)
39	MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the
40		group of the specified communicator cannot expose
41		shared memory)
42	MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called
43		function
44		
45	Table 11 9. Ennor	classes in one sided communication poutings
46	Table 11.2: Effor	classes in one-sided communication routines
47		

# 11.7 Semantics and Correctness

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

- An RMA operation is completed at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_FENCE, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the origin.
- 2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.
- 3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.
- 4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL, then the operation is completed at the target by that same call.
- 5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.
- 6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE, MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the update of a private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed in the RMA separate memory model until the process executes a suitable 

# Unofficial Draft for Comment Only

 $\overline{7}$ 

 24 

 31 

synchronization call, while they must complete in the RMA unified model without additional
 synchronization calls. If fence or post-start-complete-wait synchronization is used, updates
 to a public window copy can be delayed in both memory models until the window owner
 executes a synchronization call. When passive target synchronization is used, it is necessary
 to update the public window copy 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.

13 The behavior of some MPI RMA operations may be *undefined* in certain situations. For 14example, the result of several origin processes performing concurrent MPI_PUT operations 15to the same target location is undefined. In addition, the result of a single origin process 16performing multiple MPI_PUT operations to the same target location within the same 17access epoch is also undefined. The result at the target may have all of the data from one 18 of the MPI_PUT operations (the "last" one, in some sense), bytes from some of each of the 19operations, or something else. In MPI-2, such operations were *erroneous*. That meant that 20an MPI implementation was permitted to signal an MPI exception. Thus, user programs or 21tools that used MPI RMA could not portably permit such operations, even if the application 22code could function correctly with such an undefined result. In MPI-3, these operations are 23not erroneous, but do not have a defined behavior.

 24 Rationale. As discussed in [6], requiring operations such as overlapping puts to 25be erroneous makes it difficult to use MPI RMA to implement programming models— 26such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, 27while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any 28 implementation that enforces this rule, as it would require significant overhead. Thus, 29 relaxing this condition does not impact existing implementations or applications. (End 30 of rationale.) 31

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.

- S1. 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.
- S2. 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.

## Unofficial Draft for Comment Only

32

33

34

35

36

37

38 39

40

41

S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

*Rationale.* The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library would have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*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 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:

- U1. 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.
- U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (*End of advice to users.*)

U3. 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

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11

12

13

14

15

16

17 18

19

20

21

22

23

24 25

26

27

28 29

30

 31 

32

33

34

35

36

37

38

39 40

41

42

43

44

45

 $46 \\ 47$ 

will depend on the behavior of the implementation. Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.

- U4. 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.
- 15U5. A put or accumulate must not access a target window once a store, put, or accumulate 16update to another (overlapping) target window has started on the same location in 17 the target window and until the update completes at the target window. Conversely, 18 a store operation to a location in a window must not start once a put or accumulate 19 update to the same location in that target window has started and until the put or 20accumulate update completes at the target. 21

In the unified memory model, in the case where the window is Advice to users. in shared memory, MPI_WIN_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI_WIN_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 11.21. (End of advice to users.)

- 30 A program that violates these rules has undefined behavior. 31
  - 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 39 40 while 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 42(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 43 is posted. 44
- 45With the post-start synchronization, the target process can tell the origin process 46that its window is now ready for RMA access; with the complete-wait synchro-47 nization, the origin process can tell the target process that it has finished its 48 RMA accesses to the window.

# **Unofficial Draft for Comment Only**

4 56

7

8

9

10

11

12

13

14

22

23

 24 

2526

27

28

29

32

33

34

35

36

37

38

41

1

2

- **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 over-lapping 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-complete-wait; 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.*)

The semantics are illustrated by the following examples:

**Example 11.6** The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK and MPI_WIN_UNLOCK calls around the store to X in process B are necessary to ensure consistency between the public and private copies of the window.

		26
Process A:	Process B:	27
	window location X	28
		29
	MPI_Win_lock(EXCLUSIVE,B)	30
	store X /* local update to private copy of B */	31
	MPI_Win_unlock(B)	32
	<pre>/* now visible in public window copy */</pre>	33
		34
MPI_Barrier	MPI_Barrier	35
		36
MPI_Win_lock(EXCLUSIVE,B)		37
MPI_Get(X) /* ok, read fro	m public window */	38
MPI_Win_unlock(B)		39

**Example 11.7** In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining load/stores and multi-process synchronization. Although the following example appears correct, the compiler or hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET returning an incorrect value of X.

 $\overline{7}$ 

 24 

```
1
 Process A:
 Process B:
\mathbf{2}
 window location X
3
4
 store X /* update to private & public copy of B */
\mathbf{5}
 MPI_Barrier
 MPI_Barrier
6
 MPI_Win_lock_all
7
 MPI_Get(X) /* ok, read from window */
8
 MPI_Win_flush_local(B)
9
 /* read value in X */
10
 MPI_Win_unlock_all
11
 MPI_BARRIER provides process synchronization, but not memory synchronization. The
12
 example could potentially be made safe through the use of compiler- and hardware-specific
13
 notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The
14
 use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct
15
16
 result.
17
 Example 11.8 The following example demonstrates the reading of a memory location
18
 updated by a remote process (Rule 6) in the RMA separate memory model. Although the
19
 MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on
20
 process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is
21
 necessary to synchronize the private copy with the public copy.
22
23
 Process A:
 Process B:
24
 window location X
25
26
 MPI_Win_lock(EXCLUSIVE,B)
27
 MPI_Put(X) /* update to public window */
28
 MPI_Win_unlock(B)
29
30
 MPI_Barrier
 MPI_Barrier
31
32
 MPI_Win_lock(EXCLUSIVE,B)
33
 /* now visible in private copy of B */
34
 load X
35
 MPI_Win_unlock(B)
36
37
 Note that in this example, the barrier is not critical to the semantic correctness. The
38
 use of exclusive locks guarantees a remote process will not modify the public copy after
39
 MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation
40
 looking for changes in X on process B would be semantically correct. The barrier is required
41
 to ensure that process A performs the put operation before process B performs the load of
42
 Х.
43
 Example 11.9 Similar to Example 11.7, the following example is unsafe even in the unified
44
 model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. While
45
 Process B does not need to explicitly synchronize the public and private copies through
46
 MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the
47
 window, the scheduling of the load could result in old values of X being returned. Compiler
```

and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

Process A:	Process B: window location X
<pre>MPI_Win_lock_all MPI_Put(X) /* update to win MPI_Win_flush(B)</pre>	ndow */
MPI_Barrier	MPI_Barrier load X
MPI_Win_unlock_all	

**Example 11.10** The following example further clarifies Rule 5. MPI_WIN_LOCK and MPI_WIN_LOCK_ALL do *not* update the public copy of a window with changes to the private copy. Therefore, there is no guarantee that process A in the following sequence will see the value of X as updated by the local store by process B before the lock.

Deve e e e e A		19
Process A:	Process B:	20
	window location X	21
	stone V (thurdate to primete some of D th)	22
	store X /* update to private copy of B */	23
	MPI_Win_lock(SHARED,B)	24
MPI_Barrier	MPI_Barrier	25
		26
MPI_Win_lock(SHARED,B)		27
MPI_Get(X) /* X may be the	X before the store */	28
MPI_Win_unlock(B)		29
	MPI_Win_unlock(B)	30
	<pre>/* update on X now visible in public window */</pre>	31

The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would guarantee process A would see the updated value of X, as the public copy of the window would be explicitly synchronized with the private copy.

**Example 11.11** Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. It is *not* guaranteed that process B reads the value of X as per the local update by process A, because neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in the public window copy.

Process A:	Process B:
window location X	
window location Y	
store Y	
MPI_Win_post(A,B) /* Y vis	ible in public window */
MPI_Win_start(A)	MPI_Win_start(A)

Unofficial Draft for Comment Only

```
1
\mathbf{2}
 store X /* update to private window */
3
4
 MPI_Win_complete
 MPI_Win_complete
\mathbf{5}
 MPI_Win_wait
6
 /* update on X may not yet visible in public window */
7
8
 MPI_Barrier
 MPI_Barrier
9
10
 MPI_Win_lock(EXCLUSIVE,A)
11
 MPI_Get(X) /* may return an obsolete value */
12
 MPI_Get(Y)
13
 MPI_Win_unlock(A)
14
 To allow process B to read the value of X stored by A the local store must be replaced by
15
 a local MPI_PUT that updates the public window copy. Note that by this replacement X
16
 may become visible in the private copy of process A only after the MPI_WIN_WAIT call in
17
 process A. The update to Y made before the MPI_WIN_POST call is visible in the public
18
 window after the MPI_WIN_POST call and therefore process B will read the proper value
19
 of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START
20
 operation, and process B would still get the value stored by process A.
21
22
 Example 11.12 The following example demonstrates the interaction of general active
23
 target synchronization with local read operations with the RMA separate memory model.
24
 Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated
25
 before the load takes place.
26
27
 Process B:
 Process A:
28
 window location X
29
30
 MPI_Win_lock(EXCLUSIVE,B)
^{31}
 MPI_Put(X) /* update to public window */
32
 MPI_Win_unlock(B)
33
34
 MPI_Barrier
 MPI_Barrier
35
36
 MPI_Win_post(B)
37
 MPI_Win_start(B)
38
39
 load X /* access to private window */
40
 /* may return an obsolete value */
41
42
 MPI_Win_complete
43
 MPI_Win_wait
44
45
 To ensure that the value put by process A is read, the local load must be replaced with a
46
 local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.
47
48
```

#### 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 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. 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 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 origin processes to overlapping target memory regions.

26The default strict ordering may incur a significant performance penalty. MPI specifies 27the info key accumulate_ordering to allow relaxation of the ordering semantics when specified 28 to any window creation function. The values for this key are as follows. If set to none, 29then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 30 in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list 31 of required access orderings at the target. Allowed values in the comma-separated list 32 are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-33 after-write ordering, respectively. These indicate whether operations of the specified type 34 complete in the order they were issued. For example, raw means that any writes must 35complete at the target before subsequent reads. These ordering requirements apply only to 36 operations issued by the same origin process and targeting the same target process. The 37 default value for accumulate_ordering is rar,raw,war,waw, which implies that writes complete at 38 the target in the order in which they were issued, reads complete at the target before any 39 writes that are issued after the reads, and writes complete at the target before any reads 40 that are issued after the writes. Any subset of these four orderings can be specified. For 41 example, if only read-after-read and write-after-write ordering is required, then the value 42of the accumulate_ordering key could be set to rar, waw. The order of values is not significant. 43

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

 $45 \\ 46$ 

44

- 47
- 48

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14 15

16 17

18

19

20

21

22

23

 24 

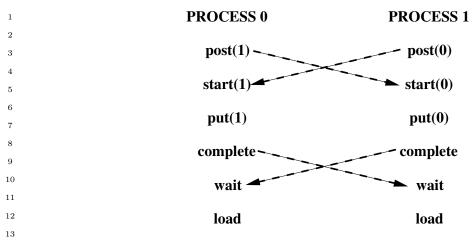


Figure 11.6: Symmetric communication

## 11.7.3 Progress

14

15 16

17

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.

21There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with 22 point-to-point communication. Access to a target window becomes enabled once the corre-23sponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On  24 25the origin process, an RMA communication may become enabled as soon as the correspond-26ing put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the 27communication must complete. 28

²⁹ Consider the code fragment in Example 11.4. Some of the calls may block if the target
 ³⁰ window is not posted. However, if the target window is posted, then the code fragment
 ³¹ must complete. The data transfer may start as soon as the put call occurs, but may be
 ³² delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 11.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed at
each process. Then, the code may deadlock, as each process may block on the start call,
waiting for the matching post to occur. Similarly, the program will deadlock if the order of
the complete and wait calls is reversed at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait

#### Unofficial Draft for Comment Only

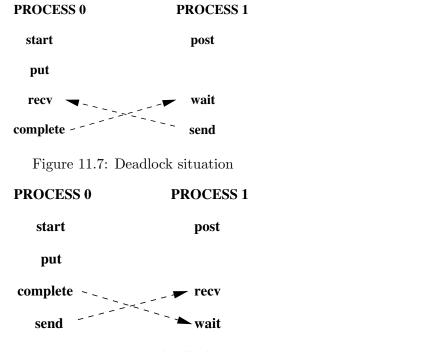


Figure 11.8: No deadlock

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 on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

*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.

A similar issue is whether such progress must occur while a process is busy comput-ing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not spec-ify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this am-biguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (End of rationale.) 

#### Unofficial Draft for Comment Only

 24 

## 11.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

The problem is illustrated by the following code:

13	Source of Process 1	Source of Process 2	Executed in Process 2
14	bbbb = 777	buff = 999	reg_A:=999
15	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
16	call MPI_PUT(bbbb		stop appl.thread
17	into buff of process 2)		buff:=777 in PUT handler
18			continue appl.thread
19	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
20		ccc = buff	ccc:=reg_A
21			

In this example, variable **buff** is allocated in the register **reg_A** and therefore **ccc** will have the old value of **buff** and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 18.1.16.

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 18.1.10–18.1.20. Sections 18.1.17 to 18.1.17 discuss several solutions for the problem in this example.

# 11.8 Examples

34 35 36

37

38

39

**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.

```
40
^{41}
 while(!converged(A)){
42
 update(A);
43
 MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
44
 for(i=0; i < toneighbors; i++)</pre>
45
 MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
46
 todisp[i], 1, totype[i], win);
47
 MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
48
 }
```

1

 $\mathbf{2}$ 

3

4 5

6

7

8

9

10

11

12

22

23

 24 

25

26

27

28

29

30

 31 

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

**Example 11.14** Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the update_core call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

**Example 11.15** Same code as in Example 11.13, rewritten using post-start-complete-wait.

**Example 11.16** Same example, with split phases, as in Example 11.14.

```
...
while(!converged(A)){
 update_boundary(A);
 MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
 MPI_Win_start(fromgroup, 0, win);
```

 $\mathbf{2}$ 

 24 

```
1
 for(i=0; i < fromneighbors; i++)</pre>
\mathbf{2}
 MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
3
 fromdisp[i], 1, fromtype[i], win);
4
 update_core(A);
5
 MPI_Win_complete(win);
6
 MPI_Win_wait(win);
7
 }
8
9
 Example 11.17 A checkerboard, or double buffer communication pattern, that allows
10
 more computation/communication overlap. Array A0 is updated using values of array A1,
11
 and vice versa. We assume that communication is symmetric: if process A gets data from
12
 process B, then process B gets data from process A. Window wini consists of array Ai.
13
14
 . . .
15
 if (!converged(A0,A1))
16
 MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
17
 MPI_Barrier(comm0);
18
 /* the barrier is needed because the start call inside the
19
 loop uses the nocheck option */
20
 while(!converged(A0, A1)){
21
 /* communication on AO and computation on A1 */
22
 update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
23
 MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
24
 for(i=0; i < fromneighbors; i++)</pre>
25
 MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
26
 fromdisp0[i], 1, fromtype0[i], win0);
27
 update1(A1); /* local update of A1 that is
28
 concurrent with communication that updates A0 */
29
 MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
30
 MPI_Win_complete(win0);
31
 MPI_Win_wait(win0);
32
33
 /* communication on A1 and computation on A0 */
34
 update2(A0, A1); /* local update of A0 that depends on A1 (and A0) */
35
 MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
36
 for(i=0; i < fromneighbors; i++)</pre>
37
 MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
38
 fromdisp1[i], 1, fromtype1[i], win1);
39
 update1(A0); /* local update of A0 that depends on A0 only,
40
 concurrent with communication that updates A1 */
41
 if (!converged(A0,A1))
42
 MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
43
 MPI_Win_complete(win1);
44
 MPI_Win_wait(win1);
45
 }
46
 A process posts the local window associated with win0 before it completes RMA accesses
47
```

492

47 A process posts the local window associated with win0 before it completes RMA accesses 48 to the remote windows associated with win1. When the wait(win1) call returns, then all neighbors of the calling process have posted the windows associated with win0. Conversely, when the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win1. Therefore, the nocheck option can be used with the calls to MPI_WIN_START.

Put calls can be used, instead of get calls, if the area of array AO (resp. A1) used by the update(A1, AO) (resp. update(AO, A1)) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

In the next several examples, for conciseness, the expression

z = MPI_Get_accumulate(...)

means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr in the description of MPI_GET_ACCUMULATE) on the left side of the assignment, in this case, z. This format is also used with MPI_COMPARE_AND_SWAP.

**Example 11.18** The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE are used to write to or read from the local public copy.

	22
Process B:	23
MPI_Win_lock_all	24
	25
	26
	27
MPI_Barrier	28
	29
MPI_Accumulate(X, MPI_SUM, -1)	30
	31
stack variable z	32
do	33
<pre>z = MPI_Get_accumulate(X,</pre>	34
MPI_NO_OP, 0)	35
MPI_Win_flush(A)	36
while(z!=0)	37
	38
MPI_Win_unlock_all	39
	40
	<pre>MPI_Win_lock_all MPI_Barrier MPI_Accumulate(X, MPI_SUM, -1) stack variable z do     z = MPI_Get_accumulate(X,         MPI_NO_OP, 0)     MPI_Win_flush(A) while(z!=0)</pre>

**Example 11.19** Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

```
1
 Process A:
 Process B:
\mathbf{2}
 window location Y
 window location X
3
 window location T
4
\mathbf{5}
 MPI_Win_lock_all
 MPI_Win_lock_all
6
 X=1
 Y=1
7
 MPI_Win_sync
 MPI_Win_sync
8
 MPI_Barrier
 MPI_Barrier
9
 MPI_Accumulate(T, MPI_REPLACE, 1)
 MPI_Accumulate(T, MPI_REPLACE, 0)
10
 stack variables t,y
 stack variable t,x
^{11}
 t=1
 t=0
12
 y=MPI_Get_accumulate(Y,
 x=MPI_Get_accumulate(X,
13
 MPI_NO_OP, 0)
 MPI_NO_OP, 0)
14
 while(y==1 && t==1) do
 while(x==1 && t==0) do
15
 y=MPI_Get_accumulate(Y,
 x=MPI_Get_accumulate(X,
16
 MPI_NO_OP, 0)
 MPI_NO_OP, 0)
17
 t=MPI_Get_accumulate(T,
 t=MPI_Get_accumulate(T,
18
 MPI_NO_OP, 0)
 MPI_NO_OP, 0)
19
 MPI_Win_flush_all
 MPI_Win_flush(A)
20
 done
 done
21
 // critical region
 // critical region
22
 MPI_Accumulate(X, MPI_REPLACE, 0)
 MPI_Accumulate(Y, MPI_REPLACE, 0)
23
 MPI_Win_unlock_all
 MPI_Win_unlock_all
^{24}
```

 Example 11.20 Implementing a critical region between multiple processes with compare and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization of A to guarantee the public copy has been updated with the initialization value found in the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure A in the public copy of Process A had been updated before the barrier.

```
Process A:
 Process B...:
32
 MPI_Win_lock_all
 MPI_Win_lock_all
33
 atomic location A
34
 A=0
35
 MPI_Win_sync
36
 MPI_Barrier
 MPI_Barrier
37
 stack variable r=1
 stack variable r=1
38
 while(r != 0) do
 while(r != 0) do
39
 r = MPI_Compare_and_swap(A, 0, 1)
 r = MPI_Compare_and_swap(A, 0, 1)
40
 MPI_Win_flush(A)
 MPI_Win_flush(A)
41
 done
 done
42
 // critical region
 // critical region
43
 r = MPI_Compare_and_swap(A, 1, 0)
 r = MPI_Compare_and_swap(A, 1, 0)
44
 MPI_Win_unlock_all
 MPI_Win_unlock_all
45
46
```

Example 11.21 The following example demonstrates the proper synchronization in the
 unified memory model when a data transfer is implemented with load and store in the

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

Process A

case of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if MPI_ASYNC_PROTECTS_NONBLOCKING is .FALSE. or the variable X is not declared as ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)

The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI_WIN_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI_WIN_SYNC operation performed by process B ensures that process A's updates to X are visible to process B.

Process B

		14
		15
MPI_WIN_LOCK_ALL(	MPI_WIN_LOCK_ALL(	16
MPI_MODE_NOCHECK,win)	MPI_MODE_NOCHECK,win)	17
		18
DO	DO	19
X=		20
		21
MPI_F_SYNC_REG(X)		22
MPI_WIN_SYNC(win)		23
MPI_SEND	MPI_RECV	24
	MPI_WIN_SYNC(win)	25
	MPI_F_SYNC_REG(X)	26
		27
	print X	28
		29
	MPI_F_SYNC_REG(X)	30
MPI_RECV	MPI_SEND	31
MPI_F_SYNC_REG(X)		32
END DO	END DO	33
		34
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)	35
		36

**Example 11.22** 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.

int	i, j;
MPI_Win	win;
MPI_Request	<pre>put_req[M] = { MPI_REQUEST_NULL };</pre>
MPI_Request	get_req;
double	*baseptr;
double	data[M][N];

 $\mathbf{2}$ 

 $\overline{7}$ 

```
1
 MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
\mathbf{2}
 MPI_COMM_WORLD, &baseptr, &win);
3
4
 MPI_Win_lock_all(0, win);
5
6
 for (i = 0; i < NSTEPS; i++) {</pre>
7
 if (i<M)
8
 j=i;
9
 else
10
 MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
11
12
 MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
13
 &get_req);
14
 MPI_Wait(&get_req,MPI_STATUS_IGNORE);
15
 compute(i, data[j], ...);
16
 MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
17
 &put_req[j]);
18
 }
19
20
 MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
21
 MPI_Win_unlock_all(win);
22
23
 Example 11.23 The following example constructs a distributed shared linked list using
24
 dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
25
 and broadcasts the pointer to all processes. All processes then concurrently append N new
26
 elements to the list. When a process attempts to attach its element to the tail of the
27
 list it may discover that its tail pointer is stale and it must chase ahead to the new tail
28
 before the element can be attached. This example requires some modification to work in
29
 an environment where the layout of the structures is different on different processes.
30
^{31}
 . . .
32
 #define NUM_ELEMS 10
33
34
 #define LLIST_ELEM_NEXT_RANK (offsetof(llist_elem_t, next) + \
35
 offsetof(llist_ptr_t, rank))
36
 #define LLIST_ELEM_NEXT_DISP (offsetof(llist_elem_t, next) + \
37
 offsetof(llist_ptr_t, disp))
38
39
 /* Linked list pointer */
40
 typedef struct {
41
 MPI_Aint disp;
42
 int
 rank;
43
 } llist_ptr_t;
44
45
 /* Linked list element */
46
 typedef struct {
47
 llist_ptr_t next;
48
```

```
1
 int value;
 \mathbf{2}
} llist_elem_t;
 3
 4
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
 5
 6
/* List of locally allocated list elements. */
 7
static llist_elem_t **my_elems = NULL;
 8
static int my_elems_size = 0;
static int my_elems_count = 0;
 9
 10
 11
/* Allocate a new shared linked list element */
MPI_Aint alloc_elem(int value, MPI_Win win) {
 12
 13
 MPI_Aint disp;
 llist_elem_t *elem_ptr;
 14
 15
 16
 /* Allocate the new element and register it with the window */
 17
 MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
 18
 elem_ptr->value = value;
 19
 elem_ptr->next = nil;
 20
 MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
 21
 /* Add the element to the list of local elements so we can free
 22
 23
 it later. */
 24
 if (my_elems_size == my_elems_count) {
 25
 my_elems_size += 100;
 26
 my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
 }
 27
 my_elems[my_elems_count] = elem_ptr;
 28
 29
 my_elems_count++;
 30
 31
 MPI_Get_address(elem_ptr, &disp);
 32
 return disp;
 33
}
 34
int main(int argc, char *argv[]) {
 35
 36
 int
 procid, nproc, i;
 37
 MPI_Win
 llist_win;
 38
 llist_ptr_t
 head_ptr, tail_ptr;
 39
 MPI_Init(&argc, &argv);
 40
 41
 42
 MPI_Comm_rank(MPI_COMM_WORLD, &procid);
 MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 43
 44
 MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
 45
 46
 47
 /* Process 0 creates the head node */
 48
 if (procid == 0)
```

```
1
 head_ptr.disp = alloc_elem(-1, llist_win);
2
3
 /* Broadcast the head pointer to everyone */
4
 head_ptr.rank = 0;
5
 MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
6
 tail_ptr = head_ptr;
7
8
 /* Lock the window for shared access to all targets */
9
 MPI_Win_lock_all(0, llist_win);
10
11
 /* All processes concurrently append NUM_ELEMS elements to the list */
12
 for (i = 0; i < NUM_ELEMS; i++) {</pre>
13
 llist_ptr_t new_elem_ptr;
14
 int success;
15
16
 /* Create a new list element and attach it to the window */
17
 new_elem_ptr.rank = procid;
18
 new_elem_ptr.disp = alloc_elem(procid, llist_win);
19
20
 /* Append the new node to the list. This might take multiple
21
 attempts if others have already appended and our tail pointer
22
 is stale. */
23
 do {
24
 llist_ptr_t next_tail_ptr = nil;
25
26
 MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
27
 (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
28
 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
29
 llist_win);
30
31
 MPI_Win_flush(tail_ptr.rank, llist_win);
32
 success = (next_tail_ptr.rank == nil.rank);
33
34
 if (success) {
35
 MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
36
 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
37
 MPI_AINT, MPI_REPLACE, llist_win);
38
39
 MPI_Win_flush(tail_ptr.rank, llist_win);
40
 tail_ptr = new_elem_ptr;
41
42
 } else {
43
 /* Tail pointer is stale, fetch the displacement. May take
44
 multiple tries if it is being updated. */
45
 do {
 MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
46
47
 1, MPI_AINT, tail_ptr.rank,
48
 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
```

Unofficial Draft for Comment Only

```
1
 1, MPI_AINT, MPI_NO_OP, llist_win);
 \mathbf{2}
 3
 MPI_Win_flush(tail_ptr.rank, llist_win);
 4
 } while (next_tail_ptr.disp == nil.disp);
 5
 tail_ptr = next_tail_ptr;
 }
 6
 7
 } while (!success);
 }
 8
 9
 10
 MPI_Win_unlock_all(llist_win);
 MPI_Barrier(MPI_COMM_WORLD);
 11
 12
 /* Free all the elements in the list */
 13
 for (; my_elems_count > 0; my_elems_count--) {
 14
 MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
 15
 16
 MPI_Free_mem(my_elems[my_elems_count-1]);
 17
 }
 18
 MPI_Win_free(&llist_win);
 19
. . .
 20
 21
 22
 23
 ^{24}
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 ^{41}
 42
 43
 44
 45
 46
 47
 48
```

1			
2			
3			
4			
5			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24 25			
25 26			
20			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42 43			
43 44			
44 45			
46			
47			
48			
-			

# Chapter 12

# **External Interfaces**

# 12.1 Introduction

This chapter begins with calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. 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 8 9	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 user. A new generalized request is started with					
11	MPI_GRE	QUEST_START(query	_fn, 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 22	OUT	request	generalized request (handle)			
23 24 25 26	<pre>int MPI_Grequest_start(MPI_Grequest_query_function *query_fn, MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state, MPI_Request *request)</pre>					
27 28 29 30 31 32 33 34 35 36	<pre>MFI_Grequest_start(query_IN, free_IN, cancel_IN, extra_state, request, ierror) PROCEDURE(MPI_Grequest_query_function) :: query_fn PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
30 37 38 39 40 41	MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR) INTEGER REQUEST, IERROR EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE					
42 43 44 45 46 47	<ul><li>Advice to users. Note that a generalized request is of the same type as regular requests, in C and Fortran. (End of advice to users.)</li><li>The call starts a generalized request and returns a handle to it in request. The syntax and meaning of the callback functions are listed below. All callback functions are passed the extra_state argument that was associated with the request by the</li></ul>					
48						

starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined	1
state for the request.	2
In C, the query function is	3
<pre>typedef int MPI_Grequest_query_function(void *extra_state,</pre>	4
MPI_Status *status);	5
in Fortran with the mpi_f08 module	6
ABSTRACT INTERFACE	7
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)	8 9
TYPE(MPI_Status) :: status	9 10
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	11
INTEGER :: ierror	12
in Fourtman with the mui module and muif h	13
in Fortran with the mpi module and mpif.h	14
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR	15
INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE	16
	17
The query_fn function computes the status that should be returned for the generalized	18
request. The status also includes information about successful/unsuccessful cancellation of	19
the request (result to be returned by MPI_TEST_CANCELLED).	20
The query_fn callback is invoked by the $MPI_{WAIT TEST}{ANY SOME ALL}$ call that	21
completed the generalized request associated with this callback. The callback function is	22
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when	23
the call occurs. In both cases, the callback is passed a reference to the corresponding	24
status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or	25
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI	26
will pass a valid status object to query_fn, and this status will be ignored upon return of the	27
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE	28
is called on the request; it may be invoked several times for the same generalized request,	29 30
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also	31
that a call to MPI_{WAIT TEST}{SOME ALL} may cause multiple invocations of query_fn	32
callback functions, one for each generalized request that is completed by the MPI call. The	33
order of these invocations is not specified by MPI.	34
In C, the free function is	35
<pre>typedef int MPI_Grequest_free_function(void *extra_state);</pre>	36
in Fortran with the mpi_f08 module	37
ABSTRACT INTERFACE	38
SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)	39
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	40
INTEGER :: ierror	41
in Fourtman with the mui module and muif h	42
in Fortran with the mpi module and mpif.h	43
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR) INTEGER IERROR	44
INTEGER TERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	45
	46 47
	47
	10

1 The free_fn function is invoked to clean up user-allocated resources when the generalized  $\mathbf{2}$ request is freed. 3 The free_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that 4 completed the generalized request associated with this callback. free_fn is invoked after  $\mathbf{5}$ the call to query_fn for the same request. However, if the MPI call completed multiple 6 generalized requests, the order in which free_fn callback functions are invoked is not specified 7by MPI. 8 The free_fn callback is also invoked for generalized requests that are freed by a call 9 to MPI_REQUEST_FREE (no call to MPI_{WAIT|TEST}{ANY|SOME|ALL} will occur for 10 such a request). In this case, the callback function will be called either in the MPI call 11MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), 12whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 13calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request 14is not deallocated until after free_fn completes. Note that free_fn will be invoked only once 15per request by a correct program. 16Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle 17 to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer 18 valid. However, user copies of this handle are valid until after free_fn completes since 19 MPI does not deallocate the object until then. Since free_fn is not called until after 20MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this 21call. Users should note that MPI will deallocate the object after free_fn executes. At 22 this point, user copies of the request handle no longer point to a valid request. MPI will 23not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid 24accessing this stale handle. This is a special case in which MPI defers deallocating the 25object until a later time that is known by the user. (End of advice to users.) 2627In C, the cancel function is 28typedef int MPI_Grequest_cancel_function(void *extra_state, int complete); 2930 in Fortran with the mpi_f08 module 31ABSTRACT INTERFACE 32 SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror) 33 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 34 LOGICAL :: complete 35INTEGER :: ierror 36 in Fortran with the mpi module and mpif.h 37 SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) 38 INTEGER IERROR 39 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 40 LOGICAL COMPLETE 41 42The cancel_fn function is invoked to start the cancelation of a generalized request. 43It is called by MPI_CANCEL(request). MPI passes complete=true to the callback function 44if MPI_GREQUEST_COMPLETE was already called on the request, and 45complete=false otherwise. 46All callback functions return an error code. The code is passed back and dealt with as 47appropriate for the error code by the MPI function that invoked the callback function. For 48

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 3 return the error code returned by the last callback, namely free_fn. If one or more of the 4 requests in a call to MPI_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return 5MPI_ERR_IN_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request 6  $\overline{7}$ 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 9 codes returned by each callback functions will be lost.

Advice to users. query_fn must not set the error field of status since query_fn may be called by MPI_WAIT or MPI_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (End of advice to users.)

MPI_GREQUEST_COMPLETE(request)					
INOUT r	request	generalized request (handle)			
<pre>int MPI_Grequest_complete(MPI_Request request)</pre>					
<pre>MPI_Grequest_complete(request, ierror)     TYPE(MPI_Request), INTENT(IN) :: request     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
	ST_COMPLETE(REQUEST, IER R 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, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI_TEST, MPI_REQUEST_FREE, or MPI_CANCEL still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions query_fn, free_fn, or cancel_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

A call to MPI_GREQUEST_COMPLETE may unblock a Advice to implementors. blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (End of advice to implementors.)

1

 $\mathbf{2}$ 

8

10

11

12

13

14

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43 44

45

46

4748

```
12.2.1 Examples
```

**Example 12.1** This example shows the code for a user-defined reduce operation on an int using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

```
7
 typedef struct {
8
 MPI_Comm comm;
9
 int tag;
10
 int root;
11
 int valin;
12
 int *valout;
13
 MPI_Request request;
14
 } ARGS;
15
16
17
 int myreduce(MPI_Comm comm, int tag, int root,
18
 int valin, int *valout, MPI_Request *request)
19
 {
20
 ARGS *args;
21
 pthread_t thread;
22
23
 /* start request */
^{24}
 MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
25
26
 args = (ARGS*)malloc(sizeof(ARGS));
27
 args->comm = comm;
28
 args->tag = tag;
29
 args->root = root;
30
 args->valin = valin;
^{31}
 args->valout = valout;
32
 args->request = *request;
33
34
 /* spawn thread to handle request */
35
 /* The availability of the pthread_create call is system dependent */
36
 pthread_create(&thread, NULL, reduce_thread, args);
37
38
 return MPI_SUCCESS;
39
 }
40
41
 /* thread code */
42
 void* reduce_thread(void *ptr)
43
 ſ
44
 int lchild, rchild, parent, lval, rval, val;
45
 MPI_Request req[2];
46
 ARGS *args;
47
48
 args = (ARGS*)ptr;
```

1

2 3

4

5

```
2
 /* compute left and right child and parent in tree; set
 3
 to MPI_PROC_NULL if does not exist */
 /* code not shown */
 4
 5
 . . .
 6
 7
 MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
 8
 MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
 MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
 9
 10
 val = lval + args->valin + rval;
 11
 MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
 if (parent == MPI_PROC_NULL) *(args->valout) = val;
 12
 MPI_Grequest_complete((args->request));
 13
 14
 free(ptr);
 15
 return(NULL);
}
 16
 17
 18
int query_fn(void *extra_state, MPI_Status *status)
 19
Ł
 /* always send just one int */
 20
 21
 MPI_Status_set_elements(status, MPI_INT, 1);
 /* can never cancel so always true */
 22
 23
 MPI_Status_set_cancelled(status, 0);
 24
 /* choose not to return a value for this */
 25
 status->MPI_SOURCE = MPI_UNDEFINED;
 26
 /* tag has no meaning for this generalized request */
 status->MPI_TAG = MPI_UNDEFINED;
 27
 /* this generalized request never fails */
 28
 29
 return MPI_SUCCESS;
}
 30
 31
 32
 33
int free_fn(void *extra_state)
 34
{
 /* this generalized request does not need to do any freeing */
 35
 /* as a result it never fails here */
 36
 37
 return MPI_SUCCESS;
}
 38
 39
 40
 41
int cancel_fn(void *extra_state, int complete)
 42
{
 /* This generalized request does not support cancelling.
 43
 44
 Abort if not already done. If done then treat as if cancel failed.*/
 45
 if (!complete) {
 46
 fprintf(stderr,
 47
 "Cannot cancel generalized request - aborting program\n");
 48
 MPI_Abort(MPI_COMM_WORLD, 99);
```

```
}
return MPI_SUCCESS;
}
```

# 12.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls to use the same request mechanism, which allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in the status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

```
25
26
```

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

```
27
 INOUT
 status
 status with which to associate count (Status)
28
 IN
 datatype
 datatype associated with count (handle)
29
30
 IN
 count
 number of elements to associate with status (integer)
^{31}
32
 int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
33
 int count)
34
 MPI_Status_set_elements(status, datatype, count, ierror)
35
 TYPE(MPI_Status), INTENT(INOUT) :: status
36
 TYPE(MPI_Datatype), INTENT(IN) ::
 datatype
37
 INTEGER, INTENT(IN) :: count
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
41
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
42
43
44
45
46
47
48
```

1

 $\mathbf{2}$ 

3

4 5

> 6 7

8

9

10

11

MPI_STATUS_SET_ELEMENTS_X(status, datatype, count) 1 2 INOUT status with which to associate count (Status) status 3 IN datatype datatype associated with count (handle) 4 IN count number of elements to associate with status (integer) 56 7 int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype, 8 MPI_Count count) 9 MPI_Status_set_elements_x(status, datatype, count, ierror) 10 TYPE(MPI_Status), INTENT(INOUT) :: status 11 TYPE(MPI_Datatype), INTENT(IN) :: datatype 12INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) :: count 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) 16INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR 17 INTEGER (KIND=MPI_COUNT_KIND) COUNT 18 These functions modify the opaque part of status so that a call to 19 MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X will return count. MPI_GET_COUNT 20will return a compatible value. 2122 The number of elements is set instead of the count because the former Rationale. 23 can deal with a nonintegral number of datatypes. (End of rationale.) 2425A subsequent call to MPI_GET_COUNT(status, datatype, count), 26MPI_GET_ELEMENTS(status, datatype, count), or 27MPI_GET_ELEMENTS_X(status, datatype, count) must use a datatype argument that has 28the same type signature as the datatype argument that was used in the call to 29 MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X. 30 31Rationale. The requirement of matching type signatures for these calls is similar 32 to the restriction that holds when **count** is set by a receive operation: in that case, 33 the calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X 34 must use a datatype with the same signature as the datatype used in the receive call. 35(End of rationale.) 36 37 38 MPI_STATUS_SET_CANCELLED(status, flag) 39 INOUT status status with which to associate cancel flag (Status) 40 41 IN flag if true indicates request was cancelled (logical) 4243 int MPI_Status_set_cancelled(MPI_Status *status, int flag) 4445MPI_Status_set_cancelled(status, flag, ierror) TYPE(MPI_Status), INTENT(INOUT) :: status 4647LOGICAL, INTENT(IN) :: flag 48 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR 3 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.

Users are advised not to reuse the status fields for values other Advice to users. 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.)

151617

18

32

33

39

40

41

42

43

44

#### 12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists 19minimal requirements for thread compliant MPI implementations and defines functions 20that can be used for initializing the thread environment. MPI may be implemented in 21environments where threads are not supported or perform poorly. Therefore, MPI imple-22 mentations are not required to be thread compliant as defined in this section. Regard-23less of whether or not the MPI implementation is thread compliant, MPI_INITIALIZED,  24 MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION 25and MPI_GET_LIBRARY_VERSION must always be thread-safe. When a thread is exe-26cuting one of these routines, if another concurrently running thread also makes an MPI call, 27the outcome will be as if the calls executed in some order. 28

This section generally assumes a thread package similar to POSIX threads [39], but the 29 syntax and semantics of thread calls are not specified here — these are beyond the scope 30 of this document.  31 

12.4.1 General

34In a thread-compliant implementation, an MPI process is a process that may be multi-35 threaded. Each thread can issue MPI calls; however, threads are not separately addressable: 36 a rank in a send or receive call identifies a process, not a thread. A message sent to a process 37 can be received by any thread in this process. 38

- This model corresponds to the POSIX model of interprocess communi-Rationale. cation: 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.)
- 45Advice to users. It is the user's responsibility to prevent races when threads within 46the same application post conflicting communication calls. The user can make sure 47 that two threads in the same process will not issue conflicting communication calls by 48 using distinct communicators at each thread. (End of advice to users.)

# **Unofficial Draft for Comment Only**

56

1

2

4

7

8

9

10

11

12

13

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

29

31

32 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. and 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 and file handles cannot affect one another. (End of rationale.)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (End of advice to implementors.)

**Exception handlers** An exception handler does not necessarily execute in the context of the 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.*)

- 4546
- 47
- 48

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
in required	desired level of thread support (integer)	29
OUT provided	provided level of thread support (integer)	30
		31
<pre>int MPI_Init_thread(int *argc, ch</pre>	ar ***argv, int required, int *provided)	32
MDT To it there does not and a second in		33
MPI_Init_thread(required, provide	-	34
INTEGER, INTENT(IN) :: requi		35
INTEGER, INTENT(OUT) :: prov		36
INTEGER, OPTIONAL, INTENT(OUT	) :: ierror	37
MPI_INIT_THREAD(REQUIRED, PROVIDE	D, IERROR)	38
INTEGER REQUIRED, PROVIDED, I	ERROR	39
		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, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 24 

 $44 \\ 45$ 

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 516). 56 **MPI_THREAD_SERIALIZED** The process may be multi-threaded, and multiple threads may  $\overline{7}$ 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 35of 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- 44 spective 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.

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)

OUT	provided	provided level of thread support (integer)
int MPI_Qu	ery_thread(int *provided)	)
INTEGE	thread(provided, ierror) R, INTENT(OUT) :: provid R, OPTIONAL, INTENT(OUT)	
- • -	THREAD(PROVIDED, IERROR) R PROVIDED, IERROR	

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().

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 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)
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 [47], collective buffering [7, 15, 48, 52, 58], and disk-directed I/O [43]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

#### 13.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 24 

filetype A *filetype* is the basis for partitioning a file among processes and defines a template
 for accessing the file. A filetype is either a single etype or a derived MPI datatype
 constructed from multiple instances of the same etype. In addition, the extent of any
 hole in the filetype must be a multiple of the etype's extent. The displacements in the
 typemap of the filetype are not required to be distinct, but they must be non-negative
 and monotonically nondecreasing.

view A view defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 13.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

offset An offset is a position in the file relative to the current view, expressed as a count of
 etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0
 is the location of the first etype visible in the view (after skipping the displacement and
 any initial holes in the view). For example, an offset of 2 for process 1 in Figure 13.2 is
 the position of the eighth etype in the file after the displacement. An "explicit offset"
 is an offset that is used as an argument in explicit data access routines.

# 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
	file a second		22
IN	filename	name of file to open (string)	23
IN	amode	file access mode (integer)	24
IN	info	info object (handle)	25
	£h.	(1 - 1)	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) 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). A process can open a file independently of

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

other processes by using the MPI_COMM_SELF communicator. The file handle returned, fh,
 can be subsequently used to access the file until the file is closed using MPI_FILE_CLOSE.
 Before calling MPI_FINALIZE, the user is required to close (via MPI_FILE_CLOSE) all files
 that were opened with MPI_FILE_OPEN. Note that the communicator comm is unaffected
 by MPI_FILE_OPEN and continues to be usable in all MPI routines (e.g., MPI_SEND).
 Furthermore, the use of comm will not interfere with I/O behavior.

The format for specifying the file name in the filename argument is implementation
 dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End of advice to implementors.)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI_FILE_SET_VIEW routine.

The following access modes are supported (specified in **amode**, a bit vector OR of the following integer constants):

- MPI_MODE_RDONLY read only,
- MPI_MODE_RDWR reading and writing,
- MPI_MODE_WRONLY write only,
- MPI_MODE_CREATE create the file if it does not exist,
- MPI_MODE_EXCL error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE delete file on close,
- MPI_MODE_UNIQUE_OPEN file will not be concurrently opened elsewhere,
  - MPI_MODE_SEQUENTIAL file will only be accessed sequentially,
  - MPI_MODE_APPEND set initial position of all file pointers to end of file.

Advice to users. 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.)

# Unofficial Draft for Comment Only

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). 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). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

13.2.2 Closing a File

MPI_FILE	_CLOSE(fh)
INOUT	fh

file handle (handle)

 $\mathbf{2}$ 

 $\overline{7}$ 

 31 

```
1
 int MPI_File_close(MPI_File *fh)
\mathbf{2}
 MPI_File_close(fh, ierror)
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)
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). The constant MPI_INFO_NULL refers to the null info, and can be used
45
 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).

13.2.4 Resizing a File		
MPI_FILE_SET_SIZE(fh, size)		
INOUT fh	file handle (handle)	
IN size	size to truncate or expand file (integer)	
<pre>int MPI_File_set_size(MPI_File fh, MPI_Offset size) MPI_File_set_size(fh, size, ierror) </pre>		
TYPE(MPI_File), INTENT(IN) :: INTEGER(KIND=MPI_OFFSET_KIND), INTEGER, OPTIONAL, INTENT(OUT)	INTENT(IN) :: size	
MPI_FILE_SET_SIZE(FH, SIZE, IERROR INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND)		

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).

 24 

```
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)
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)
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
```

524

1 INTEGER(KIND=MPI_OFFSET_KIND) SIZE  $\mathbf{2}$ MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with 3 the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a 4 data access operation (see Section 13.6.1). 56 13.2.7 Querying File Parameters 7 8 9 MPI_FILE_GET_GROUP(fh, group) 10 11 IN fh file handle (handle) 12OUT group which opened the file (handle) group 13 14int MPI_File_get_group(MPI_File fh, MPI_Group *group) 1516MPI_File_get_group(fh, group, ierror) 17 TYPE(MPI_File), INTENT(IN) :: fh 18 TYPE(MPI_Group), INTENT(OUT) :: group 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI_FILE_GET_GROUP(FH, GROUP, IERROR) 21INTEGER FH, GROUP, IERROR 22 23MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to  24 open the file associated with fh. The group is returned in group. The user is responsible for 25freeing group. 2627MPI_FILE_GET_AMODE(fh, amode) 2829 IN fh file handle (handle) 30 OUT amode file access mode used to open the file (integer) 3132 33 int MPI_File_get_amode(MPI_File fh, int *amode) 34 MPI_File_get_amode(fh, amode, ierror) 35 TYPE(MPI_File), INTENT(IN) :: fh 36 INTEGER, INTENT(OUT) :: amode 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI_FILE_GET_AMODE(FH, AMODE, IERROR) INTEGER FH, AMODE, IERROR 40 41 MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with 42fh. 43 44**Example 13.1** In Fortran 77, decoding an **amode** bit vector will require a routine such as 45the following: 46

```
1
 SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
\mathbf{2}
 ļ
3
 !
 TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
4
 !
 IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
5
 Ţ
6
 INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
7
 BIT_FOUND = 0
8
 CP_AMODE = AMODE
9
 100 CONTINUE
10
 LBIT = 0
11
 HIFOUND = 0
12
 DO 20 L = MAX_BIT, 0, -1
13
 MATCHER = 2**L
14
 IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
15
 HIFOUND = 1
16
 LBIT = MATCHER
17
 CP_AMODE = CP_AMODE - MATCHER
18
 END IF
19
 20
 CONTINUE
20
 IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
21
 IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
22
 CP_AMODE .GT. 0) GO TO 100
23
 END
24
 This routine could be called successively to decode amode, one bit at a time. For
25
 example, the following code fragment would check for MPI_MODE_RDONLY.
26
27
 CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
 IF (BIT_FOUND .EQ. 1) THEN
28
 PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
29
30
 ELSE
 PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
31
32
 END IF
33
34
 13.2.8 File Info
35
```

Hints specified via info (see Chapter 9) allow a user to provide information such as file 36 access patterns and file system specifics to direct optimization. Providing hints may enable 37 an implementation to deliver increased I/O performance or minimize the use of system 38 resources. An implementation is free to ignore all hints; however, applications must comply 39 with any info hints they provide that are used by the MPI implementation (i.e., are returned 40 by a call to MPI_FILE_GET_INFO) and that place a restriction on the behavior of the 41 application. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, 42MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object. When an info 43 object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or 44 MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the 45 info does not specify. 46

47Advice to implementors. It may happen that a program is coded with hints for one 48 system, and later executes on another system that does not support these hints. In

```
MPI_FILE_SET_INFO(fh, info)
 INOUT
 fh
 file handle (handle)
 IN
 info
 info object (handle)
int MPI_File_set_info(MPI_File fh, MPI_Info info)
MPI_File_set_info(fh, info, ierror)
 TYPE(MPI_File), INTENT(IN) ::
 fh
 TYPE(MPI_Info), INTENT(IN) ::
 info
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
MPI_FILE_SET_INFO(FH, INFO, IERROR)
 INTEGER FH, INFO, IERROR
```

MPI_FILE_SET_INFO updates the hints of the file associated with fh using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_FILE_SET_INFO. MPI_FILE_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. An implementation may also be unable to update certain info hints in a call to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO. MPI_FILE_GET_INFO can be used to determine whether info changes were ignored by the implementation. (*End of advice to users.*)

MPI_FILE_GET_INFO(fh, info_used)

IN	fh	file handle (handle)	40
OUT	info_used	new info object (handle)	41
	_		42
int MDT E	ile_get_info(MPI_File fh,	MDI Info tinfo used)	43
IIIC MFI_F	IIe_get_IIIO(MFI_FIIe III,	MF1_1110 *1110_dsed)	44
MPI_File_	get_info(fh, info_used, i	error)	45
TYPE (	<pre>MPI_File), INTENT(IN) ::</pre>	fh	46
TYPE (	<pre>MPI_Info), INTENT(OUT) ::</pre>	info_used	47
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	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

19

20 21

22

23

 24 

25

26

27

28

29

30

 31 

32

33

34

1	MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
2	INTEGER FH, INFO_USED, IERROR
3	
4	MPI_FILE_GET_INFO returns a new info object containing the hints of the file associ-
5	ated with fh. The current setting of all hints related to this file is returned in info_used. An
6	MPI implementation is required to return all hints that are supported by the implementa-
7	tion and have default values specified; any user-supplied hints that were not ignored by the
8	implementation; and any additional hints that were set by the implementation. If no such
9	hints exist, a handle to a newly created info object is returned that contains no key/value
10	pairs. The user is responsible for freeing info_used via MPI_INFO_FREE.
11	
12	Reserved File Hints
13	
14	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
15	interpret the key value, it must provide the functionality described. (For more details on
16	"info," see Chapter 9.)
17	These hints mainly affect access patterns and the layout of data on parallel I/O devices.
18	For each hint name introduced, we describe the purpose of the hint, and the type of the hint
19	value. The " $[SAME]$ " annotation specifies that the hint values provided by all participating
20	processes must be identical; otherwise the program is erroneous. In addition, some hints are
21	context dependent, and are only used by an implementation at specific times (e.g., file_perm
22	is only useful during file creation).
23	
24	access_style (comma separated list of strings): This hint specifies the manner in which
25	the file will be accessed until the file is closed or until the access_style key value is
26	altered. The hint value is a comma separated list of the following: read_once, write_once,
27	read_mostly, write_mostly, sequential, reverse_sequential, ${ m and}$ random.
28	collective_buffering (boolean) [SAME]: This hint specifies whether the application may
29	benefit from collective buffering. Collective buffering is an optimization performed
30	on collective accesses. Accesses to the file are performed on behalf of all processes in
31	· · ·
32	the group by a number of target nodes. These target nodes coalesce small requests
33	into large disk accesses. Valid values for this key are true and false. Collective buffering
34	parameters are further directed via additional hints: cb_block_size, cb_buffer_size, and
35	cb_nodes.
36	cb_block_size (integer) [SAME]: This hint specifies the block size to be used for collective
37	buffering file access. <i>Target nodes</i> access data in chunks of this size. The chunks are
38	distributed among target nodes in a round-robin (cyclic) pattern.
39	distributed among target nodes in a round room (cyche) pattern.
40	cb_buffer_size (integer) [SAME]: This hint specifies the total buffer space that can be used
41	for collective buffering on each target node, usually a multiple of cb_block_size.
42	
43	cb_nodes (integer) [SAME]: This hint specifies the number of target nodes to be used for
44	collective buffering.
45	chunked (comma separated list of integers) [SAME]: This hint specifies that the file
46	consists of a multidimentional array that is often accessed by subarrays. The value
47	for this hint is a comma separated list of array dimensions, starting from the most
48	significant one (for an array stored in row-major order, as in C, the most significant

dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

- chunked_item (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- chunked_size (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- filename (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.
- file_perm (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode that includes MPI_MODE_CREATE. The set of valid values for this key is implementation dependent.
- io_node_list (comma separated list of strings) [SAME]: This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.
- nb_proc (integer) [SAME]: This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.
- 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.

 $45 \\ 46$ 

 $\overline{7}$ 

 24 

 31 

```
530
 CHAPTER 13. I/O
 13.3
 File Views
1
\mathbf{2}
3
4
 MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
5
 INOUT
 file handle (handle)
6
 fh
7
 IN
 disp
 displacement (integer)
8
 IN
 elementary datatype (handle)
 etype
9
 IN
 filetype
 filetype (handle)
10
11
 IN
 datarep
 data representation (string)
12
 IN
 info
 info object (handle)
13
14
 int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
15
 MPI_Datatype filetype, const char *datarep, MPI_Info info)
16
17
 MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
20
 TYPE(MPI_Datatype), INTENT(IN) ::
 etype, filetype
21
 CHARACTER(LEN=*), INTENT(IN) :: datarep
22
 TYPE(MPI_Info), INTENT(IN) :: info
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
 MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
25
 INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
26
 CHARACTER*(*) DATAREP
27
 INTEGER(KIND=MPI_OFFSET_KIND) DISP
28
29
 The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file.
30
 The start of the view is set to disp; the type of data is set to etype; the distribution of data
^{31}
 to processes is set to filetype; and the representation of data in the file is set to datarep.
32
 In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file
33
 pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents
34
 of etype in the file data representation must be identical on all processes in the group; values
35
 for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be
36
 committed.
37
 The etype always specifies the data layout in the file. If etype is a portable datatype (see
38
 Section 2.4), the extent of etype is computed by scaling any displacements in the datatype
39
 to match the file data representation. If etype is not a portable datatype, no scaling is done
40
 when computing the extent of etype. The user must be careful when using nonportable
41
 etypes in heterogeneous environments; see Section 13.5.1 for further details.
42
 If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special
43
 displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displace-
44
 ment to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is
45
 invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.
46
47
 Rationale. For some sequential files, such as those corresponding to magnetic tapes
48
 or streaming network connections, the displacement may not be meaningful.
```

MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

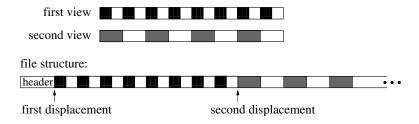


Figure 13.3: Displacements

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). (End of advice to users.)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype, and filetype arguments can be changed via future calls to MPI_FILE_SET_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype.

#### Unofficial Draft for Comment Only

 31 

 $45 \\ 46$ 

⁽End of advice to users.)

```
1
 The info argument is used to provide information regarding file access patterns and file
\mathbf{2}
 system specifics to direct optimization (see Section 13.2.8). The constant MPI_INFO_NULL
3
 refers to the null info and can be used when no info needs to be specified.
4
 The datarep argument is a string that specifies the representation of data in the file.
5
 See the file interoperability section (Section 13.5) for details and a discussion of valid values.
6
 The user is responsible for ensuring that all nonblocking requests and split collective
7
 operations on fh have been completed before calling MPI_FILE_SET_VIEW — otherwise,
8
 the call to MPI_FILE_SET_VIEW is erroneous.
9
10
 MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)
11
12
 IN
 fh
 file handle (handle)
13
 OUT
 disp
 displacement (integer)
14
 OUT
 elementary datatype (handle)
 etype
15
16
 OUT
 filetype
 filetype (handle)
17
 OUT
 datarep
 data representation (string)
18
19
 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
20
 MPI_Datatype *filetype, char *datarep)
21
22
 MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
23
 TYPE(MPI_File), INTENT(IN) :: fh
^{24}
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::
 disp
25
 TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
26
 CHARACTER(LEN=*), INTENT(OUT) :: datarep
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
 MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
29
 INTEGER FH, ETYPE, FILETYPE, IERROR
30
 CHARACTER*(*) DATAREP
^{31}
 INTEGER(KIND=MPI_OFFSET_KIND) DISP
32
33
 MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current
34
 value of the displacement is returned in disp. The etype and filetype are new datatypes with
35
 typemaps equal to the typemaps of the current etype and filetype, respectively.
36
 The data representation is returned in datarep. The user is responsible for ensuring
37
 that datarep is large enough to hold the returned data representation string. The length of
38
 a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.
39
 In addition, if a portable datatype was used to set the current view, then the corre-
40
```

⁴⁰ sponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype
 ⁴¹ or filetype are derived datatypes, the user is responsible for freeing them. The etype and
 ⁴² filetype returned are both in a committed state.

- 44
- 45
- 46
- 47
- 48

# 13.4 Data Access

### 13.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 13.1.

positioning	synchronism	coordination	
		noncollective	collective
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
offsets	-	MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT	MPI_FILE_IREAD_AT_ALL
		MPI_FILE_IWRITE_AT	MPI_FILE_IWRITE_AT_ALL
	split collective	N/A	MPI_FILE_READ_AT_ALL_BEGIN
			MPI_FILE_READ_AT_ALL_END
			MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL
	nonblocking	MPI_FILE_IREAD	MPI_FILE_IREAD_ALL
		MPI_FILE_IWRITE	MPI_FILE_IWRITE_ALL
	split collective	N/A	MPI_FILE_READ_ALL_BEGIN
			MPI_FILE_READ_ALL_END
			MPI_FILE_WRITE_ALL_BEGIN
			MPI_FILE_WRITE_ALL_END
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED
	nonblocking	MPI_FILE_IREAD_SHARED	N/A
		MPI_FILE_IWRITE_SHARED	
	split collective	N/A	MPI_FILE_READ_ORDERED_BEGIN
			MPI_FILE_READ_ORDERED_END
			MPI_FILE_WRITE_ORDERED_BEGIN
			MPI_FILE_WRITE_ORDERED_END

Table 13.1: Data access routines

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 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.

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. The data access routines that use shared file pointers contain _SHARED or _ORDERED

#### 

 $\overline{7}$ 

 31 

in their name (e.g., MPI_FILE_WRITE_SHARED). Operations with shared file pointers are described in Section 13.4.4.

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,

10 11

20

8

9

1

 $\mathbf{2}$ 

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.

# ¹⁷ Synchronism

¹⁹ MPI supports blocking and nonblocking I/O routines.

A blocking I/O call will not return until the I/O request is completed.

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$ 

A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. Given suitable hardware, this allows the transfer of data out of and into 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).

# 3334 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 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.

45 46

- 46 47
- +1 48

#### Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 4.1.11. The data is accessed from those parts of the file specified by the current view (Section 13.3). 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 18.1.10–18.1.20. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS (or MPI_GET_ELEMENTS_X), respectively. The interpretation of the MPI_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. The status can be passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other fields of status are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

### 13.4.2 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.

# 

 $\overline{7}$ 

 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)
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
 offset
 IN
 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
 TYPE(MPI_File), INTENT(IN) :: fh
44
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
45
 TYPE(*), DIMENSION(..) :: buf
46
 INTEGER, INTENT(IN) ::
 count
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
 TYPE(MPI_Status) :: status
```

INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	1
<type< td=""><td colspan="3"><pre>MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)      <type> BUF(*)</type></pre></td></type<>	<pre>MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)      <type> BUF(*)</type></pre>		
	ER FH, COUNT, DATATYPE, S ER(KIND=MPI_OFFSET_KIND)	TATUS(MPI_STATUS_SIZE), IERROR OFFSET	5 6
	ILE_READ_AT_ALL is a colle	ctive version of the blocking MPI_FILE_READ_AT	7
interface.			8 9
			10
	WRITE_AT(fh, offset, buf, cou		11 12
INOUT	fh	file handle (handle)	12
IN	offset	file offset (integer)	14
IN	buf	initial address of buffer (choice)	15
IN	count	number of elements in buffer (integer)	16
IN	datatype	datatype of each buffer element (handle)	17
OUT	status	status object (Status)	18 19
			20
int MPI_F		MPI_Offset offset, const void *buf,	21
	int count, MPI_Dataty	ype datatype, MPI_Status *status)	22
MPI_File_	write_at(fh, offset, buf,	count, datatype, status, ierror)	23
	<pre>MPI_File), INTENT(IN) ::</pre>		24 25
	ER(KIND=MPI_OFFSET_KIND),		25 26
	*), DIMENSION(), INTENT ER, INTENT(IN) :: count	(IN) :: but	27
	MPI_Datatype), INTENT(IN)	:: datatype	28
	MPI_Status) :: status		29
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	30
MPT FTIF V	WRITE AT(FH OFFSFT BUF	COUNT, DATATYPE, STATUS, IERROR)	31 32
	> BUF(*)	obowi, Daiaini, Dialob, Himon,	33
51		TATUS(MPI_STATUS_SIZE), IERROR	34
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	35
MPI_F	ILE_WRITE_AT writes a file b	beginning at the position specified by offset.	36
_			37
	MUTE AT ALL (the offerst hu	f count datatura status)	38 39
	WRITE_AT_ALL(fh, offset, bu	,	40
INOUT	fh	file handle (handle)	41
IN	offset	file offset (integer)	42
IN	buf	initial address of buffer (choice)	43
IN	count	number of elements in buffer (integer)	44
IN	datatype	datatype of each buffer element (handle)	45 46
OUT	status	status object (Status)	47
			48

```
1
 int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
\mathbf{2}
 int count, MPI_Datatype datatype, MPI_Status *status)
3
 MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
4
 TYPE(MPI_File), INTENT(IN) :: fh
5
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
6
 TYPE(*), DIMENSION(...), INTENT(IN) :: 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
12
 MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
13
 <type> BUF(*)
14
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
15
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
16
 MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
17
 MPI_FILE_WRITE_AT interface.
18
19
20
 MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
21
 IN
 fh
 file handle (handle)
22
23
 IN
 offset
 file offset (integer)
24
 OUT
 buf
 initial address of buffer (choice)
25
 IN
 count
 number of elements in buffer (integer)
26
27
 IN
 datatype
 datatype of each buffer element (handle)
28
 OUT
 request object (handle)
 request
29
30
 int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
^{31}
 MPI_Datatype datatype, MPI_Request *request)
32
33
 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
34
 TYPE(MPI_File), INTENT(IN) :: fh
35
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
36
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
37
 INTEGER, INTENT(IN) :: count
38
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
 TYPE(MPI_Request), INTENT(OUT) ::
 request
40
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
41
 MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
42
 <type> BUF(*)
43
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
44
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
46
 MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
47
48
```

MPI_FILE	_IREAD_AT_ALL(fh, offset, bu	f, count, datatype, request)	1
IN	fh	file handle (handle)	2
IN	offset	file offset (integer)	3 4
OUT	buf	initial address of buffer (choice)	5
IN	count	number of elements in buffer (integer)	6
IN	datatype	datatype of each buffer element (handle)	7
OUT			8 9
001	request	request object (handle)	10
int MPI H	int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,		
		ype datatype, MPI_Request *request)	12
MPT Filo	iread at all(fh offset	huf count datature request ierror)	13 14
<pre>MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)         TYPE(MPI_File), INTENT(IN) :: fh</pre>			
	GER(KIND=MPI_OFFSET_KIND)	, INTENT(IN) :: offset	$15 \\ 16$
	(*), DIMENSION(), ASYNCH	HRONOUS :: buf	17
	GER, INTENT(IN) :: count		18
	(MPI_Datatype), INTENT(IN) (MPI_Request), INTENT(OUT)		19
	GER, OPTIONAL, INTENT(OUT)	-	20 21
		BUF, COUNT, DATATYPE, REQUEST, IERROR)	22
	<pre>BUF(*)</pre>	DUF, CUUNI, DAIAIIFE, REQUESI, IERRUR/	23
• 1	GER FH, COUNT, DATATYPE, I	REQUEST, IERROR	24
INTEG	GER(KIND=MPI_OFFSET_KIND)	OFFSET	25
MPI_	FILE_IREAD_AT_ALL is a non	blocking version of MPI_FILE_READ_AT_ALL. See	26 27
	.6.5 for semantics of nonblock	8	28
			29
MPI_FILE	_IWRITE_AT(fh, offset, buf, cc	ount, datatype, request)	$30 \\ 31$
INOUT	fh	file handle (handle)	32
IN	offset	file offset (integer)	33
IN	buf	initial address of buffer (choice)	34
IN	count	number of elements in buffer (integer)	35
IN	datatype	datatype of each buffer element (handle)	36 37
		• -	38
OUT	request	request object (handle)	39
int MPT I	File iwrite at(MPT File fl	h, MPI_Offset offset, const void *buf,	40
1110 111 1_1		ype datatype, MPI_Request *request)	41
MDT Eila		f, count, datatype, request, ierror)	42 43
	(MPI_File), INTENT(IN) ::	VI I	44
	GER(KIND=MPI_OFFSET_KIND)		45
		I(IN), ASYNCHRONOUS :: buf	46
	GER, INTENT(IN) :: count		47 48
TYPE	(MPI_Datatype), INTENT(IN)	) :: datatype	40

```
1
 TYPE(MPI_Request), INTENT(OUT) ::
 request
\mathbf{2}
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
3
 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
 <type> BUF(*)
5
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
6
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
7
8
 MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
9
10
 MPI_FILE_IWRITE_AT_ALL(fh, offset, buf, count, datatype, request)
11
12
 INOUT
 fh
 file handle (handle)
13
 IN
 offset
 file offset (integer)
14
 IN
 buf
 initial address of buffer (choice)
15
16
 IN
 count
 number of elements in buffer (integer)
17
 IN
 datatype of each buffer element (handle)
 datatype
18
 OUT
 request
 request object (handle)
19
20
21
 int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
22
 int count, MPI_Datatype datatype, MPI_Request *request)
23
 MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
24
 TYPE(MPI_File), INTENT(IN) :: fh
25
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
26
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
27
 INTEGER, INTENT(IN) :: count
28
 TYPE(MPI_Datatype), INTENT(IN) ::
 datatype
29
 TYPE(MPI_Request), INTENT(OUT) ::
 request
30
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
^{31}
32
 MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
 <type> BUF(*)
34
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
36
 MPI_FILE_IWRITE_AT_ALL is a nonblocking version of MPI_FILE_WRITE_AT_ALL.
37
38
 13.4.3 Data Access with Individual File Pointers
39
40
 MPI maintains one individual file pointer per process per file handle. The current value
41
 of this pointer implicitly specifies the offset in the data access routines described in this
42
 section. These routines only use and update the individual file pointers maintained by MPI.
43
 The shared file pointer is not used nor updated.
44
 The individual file pointer routines have the same semantics as the data access with
45
 explicit offset routines described in Section 13.4.2, with the following modification:
46
47
 • the offset is defined to be the current value of the MPI-maintained individual file
48
 pointer.
```

the end of file is reached:

After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file. 4 If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.

MPI_FILE_READ(fh, buf, count, datatype, status)

```
9
 INOUT
 fh
 file handle (handle)
 10
 OUT
 buf
 initial address of buffer (choice)
 11
 number of elements in buffer (integer)
 IN
 count
 12
 13
 IN
 datatype of each buffer element (handle)
 datatype
 14
 OUT
 status
 status object (Status)
 15
 16
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
 17
 MPI_Status *status)
 18
 19
MPI_File_read(fh, buf, count, datatype, status, ierror)
 20
 TYPE(MPI_File), INTENT(IN) :: fh
 21
 TYPE(*), DIMENSION(...)
 ::
 buf
 22
 INTEGER, INTENT(IN) :: count
 23
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 ^{24}
 TYPE(MPI_Status) :: status
 25
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
 26
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
 27
 <type> BUF(*)
 28
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
 29
 30
 MPI_FILE_READ reads a file using the individual file pointer.
 31
Example 13.2 The following Fortran code fragment is an example of reading a file until
 32
```

! Read a preexisting input file until all data has been read. 35 Call routine "process_input" if all requested data is read. ! 36 ! The Fortran 90 "exit" statement exits the loop. 37 38 bufsize, numread, totprocessed, status(MPI_STATUS_SIZE) integer 39 parameter (bufsize=100) 40 localbuffer(bufsize) real 41 integer (kind=MPI_OFFSET_KIND) zero 4243 zero = 04445call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', & 46MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr ) 47call MPI_FILE_SET_VIEW( myfh, zero, MPI_REAL, MPI_REAL, 'native', & 48

1

2

3

56 7

8

```
1
 MPI_INFO_NULL, ierr)
2
 totprocessed = 0
3
 do
4
 call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, &
5
 status, ierr)
6
 call MPI_GET_COUNT(status, MPI_REAL, numread, ierr)
7
 call process_input(localbuffer, numread)
8
 totprocessed = totprocessed + numread
9
 if (numread < bufsize) exit
10
 enddo
11
12
 write(6,1001) numread, bufsize, totprocessed
13
 1001 format("No more data: read", I3, "and expected", I3, &
14
 "Processed total of", I6, "before terminating job.")
15
16
 call MPI_FILE_CLOSE(myfh, ierr)
17
18
19
 MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
20
21
 INOUT
 fh
 file handle (handle)
22
 OUT
 buf
 initial address of buffer (choice)
23
 IN
 count
 number of elements in buffer (integer)
24
25
 IN
 datatype
 datatype of each buffer element (handle)
26
 OUT
 status
 status object (Status)
27
28
 int MPI_File_read_all(MPI_File fh, void *buf, int count,
29
 MPI_Datatype datatype, MPI_Status *status)
30
^{31}
 MPI_File_read_all(fh, buf, count, datatype, status, ierror)
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
 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
39
 <type> BUF(*)
40
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
41
42
 MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
43
44
45
46
47
48
```

MPI_FILE_WRITE(fh, buf, count, datatype, status)			
INOUT	fh	file handle (handle)	2 3
IN	buf	initial address of buffer (choice)	4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
OUT	status	status object (Status)	7 8
			9
int MPI_F	'ile_write(MPI_File fh, co	onst void *buf, int count,	10
	MPI_Datatype datatyp	e, MPI_Status *status)	11
MPI_File_	write(fh, buf, count, dat	tatype, status, ierror)	12 13
	<pre>MPI_File), INTENT(IN) ::</pre>	fh	10
	*), DIMENSION(), INTEN	T(IN) :: buf	15
	ER, INTENT(IN) :: count MPI_Datatype), INTENT(IN)	) :: datatype	16
	MPI_Status) :: status	, dababypo	17
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	18 19
MPI FILE	WRITE(FH, BUF, COUNT, DAT	TATYPE, STATUS, IERROR)	20
	> BUF(*)	,,,	21
INTEG	ER FH, COUNT, DATATYPE, S	STATUS(MPI_STATUS_SIZE), IERROR	22
MPI_F	FILE_WRITE writes a file usin	g the individual file pointer.	23
			24 25
	_WRITE_ALL(fh, buf, count, d	atatype status)	26
	`	,	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
int MPI_F		h, const void *buf, int count,	35 36
	MPI_Datatype datatyp	e, MPI_Status *status)	37
		, datatype, status, ierror)	38
	<pre>MPI_File), INTENT(IN) ::     DIMENSION( ) INTENT</pre>		39
	*), DIMENSION(), INTEN ER, INTENT(IN) :: count		40
	MPI_Datatype), INTENT(IN)	) :: datatype	41 42
	MPI_Status) :: status		43
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	44
MPI_FILE_	WRITE_ALL(FH, BUF, COUNT	, DATATYPE, STATUS, IERROR)	45
• -	<type> BUF(*) 4</type>		
INTEG	ER FH, COUNT, DATATYPE, S	STATUS(MPI_STATUS_SIZE), IERROR	47 48
			-10

```
1
 MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
\mathbf{2}
 face.
3
4
 MPI_FILE_IREAD(fh, buf, count, datatype, request)
5
6
 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
 request object (handle)
12
13
 int MPI_File_iread(MPI_File fh, void *buf, int count,
14
 MPI_Datatype datatype, MPI_Request *request)
15
16
 MPI_File_iread(fh, buf, count, datatype, request, ierror)
17
 TYPE(MPI_File), INTENT(IN) :: fh
18
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
19
 INTEGER, INTENT(IN) :: count
20
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
 TYPE(MPI_Request), INTENT(OUT) ::
 request
22
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
23
 MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
24
 <type> BUF(*)
25
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
26
27
 MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
28
29
 Example 13.3 The following Fortran code fragment illustrates file pointer update seman-
30
 tics:
^{31}
 Read the first twenty real words in a file into two local
32
 L
 buffers. Note that when the first MPI_FILE_IREAD returns,
33
 Т
34
 !
 the file pointer has been updated to point to the
 !
 eleventh real word in the file.
35
36
 bufsize, req1, req2
37
 integer
 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
42
 zero = 0
43
 call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
44
 MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
45
 call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
46
 MPI_INFO_NULL, ierr)
47
 call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
48
```

1 req1, ierr )  $\mathbf{2}$ call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, & 3 req2, ierr ) 4 call MPI_WAIT( req1, status1, ierr ) 56 call MPI_WAIT( req2, status2, ierr ) 7 8 call MPI_FILE_CLOSE( myfh, ierr ) 9 10 11 MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request) 12INOUT file handle (handle) fh 13 14OUT buf initial address of buffer (choice) 15IN count number of elements in buffer (integer) 16IN datatype datatype of each buffer element (handle) 1718 OUT request object (handle) request 19 20int MPI_File_iread_all(MPI_File fh, void *buf, int count, 21MPI_Datatype datatype, MPI_Request *request) 22 MPI_File_iread_all(fh, buf, count, datatype, request, ierror) 23TYPE(MPI_File), INTENT(IN) :: fh 24TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 25INTEGER, INTENT(IN) :: count 26TYPE(MPI_Datatype), INTENT(IN) :: datatype 27TYPE(MPI_Request), INTENT(OUT) :: request 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 31<type> BUF(*) 32 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 33 34 MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL. 35 36 MPI_FILE_IWRITE(fh, buf, count, datatype, request) 37 INOUT 38 fh file handle (handle) 39 IN buf initial address of buffer (choice) 40 IN count number of elements in buffer (integer) 41 42IN datatype datatype of each buffer element (handle) 43 OUT request request object (handle) 4445int MPI_File_iwrite(MPI_File fh, const void *buf, int count, 46MPI_Datatype datatype, MPI_Request *request) 4748 MPI_File_iwrite(fh, buf, count, datatype, request, ierror)

```
1
 TYPE(MPI_File), INTENT(IN) :: fh
\mathbf{2}
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
3
 INTEGER, INTENT(IN) :: count
4
 TYPE(MPI_Datatype), INTENT(IN) ::
 datatype
5
 TYPE(MPI_Request), INTENT(OUT) ::
 request
6
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
7
 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
8
 <type> BUF(*)
9
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
10
11
 MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
12
13
 MPI_FILE_IWRITE_ALL(fh, buf, count, datatype, request)
14
15
 INOUT
 fh
 file handle (handle)
16
 IN
 buf
 initial address of buffer (choice)
17
 IN
 count
 number of elements in buffer (integer)
18
19
 IN
 datatype of each buffer element (handle)
 datatype
20
 OUT
 request
 request object (handle)
21
22
 int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,
23
 MPI_Datatype datatype, MPI_Request *request)
^{24}
25
 MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
26
 TYPE(MPI_File), INTENT(IN) :: fh
27
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
28
 INTEGER, INTENT(IN) :: count
29
 TYPE(MPI_Datatype), INTENT(IN) ::
 datatype
30
 TYPE(MPI_Request), INTENT(OUT) ::
 request
31
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
32
 MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
 <type> BUF(*)
34
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
36
 MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
37
38
39
 MPI_FILE_SEEK(fh, offset, whence)
40
 INOUT
 fh
 file handle (handle)
41
 IN
 offset
 file offset (integer)
42
43
 IN
 whence
 update mode (state)
44
45
 int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
46
 MPI_File_seek(fh, offset, whence, ierror)
47
 TYPE(MPI_File), INTENT(IN) ::
 fh
48
```

INTEGE	ER(KIND=MPI_OFFSET_KIND), ER, INTENT(IN) :: whence ER, OPTIONAL, INTENT(OUT)		1 2 3
INTEGE	SEEK(FH, OFFSET, WHENCE, ER FH, WHENCE, IERROR ER(KIND=MPI_OFFSET_KIND)		4 5 6 7
	ILE_SEEK updates the individ ossible values:	lual file pointer according to whence, which has the	8 9
<ul> <li>MPI_S</li> </ul>	$SEEK_SET$ : the pointer is set t	o offset	10 11
• MPI_S	SEEK_CUR: the pointer is set t	to the current pointer position plus offset	12
• MPLS	EEK_END: the pointer is set t	to the end of file plus offset	13
The of	-	lows seeking backwards. It is erroneous to seek to	14 15 16 17
MPL FILE	GET_POSITION(fh, offset)		18
IN	fh	file handle (handle)	19 20
OUT	offset	offset of individual pointer (integer)	21
		onsoo or marriaam pomoor (mooger)	22
int MPI_Fi	ile_get_position(MPI_File	e fh, MPI_Offset *offset)	23 24
MPI File a	get_position(fh, offset,	ierror)	24 25
	<pre>MPI_File), INTENT(IN) ::</pre>	fh	26
	ER(KIND=MPI_OFFSET_KIND), ER, OPTIONAL, INTENT(OUT)		27 28
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET			29 30 31
			32
_	ILE_GET_POSITION returns, type units relative to the cur	in offset, the current position of the individual file	$33 \\ 34$
Advice to users. The offset can be used in a future call to MPI_FILE_SEEK 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. ( <i>End of advice to users.</i> )			35 36 37 38 39 40 41
MPI_FILE_	GET_BYTE_OFFSET(fh, offse	et, disp)	42 43
IN	fh	file handle (handle)	44
IN	offset	offset (integer)	45
OUT	disp	absolute byte position of offset (integer)	46
		associate of the bootering of output (model)	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)    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	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.
23 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, with the following modifications:
26	$\bullet$ 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  $\mathbf{2}$ 4 MPI_FILE_READ_SHARED(fh, buf, count, datatype, status) 5INOUT fh file handle (handle) 6 OUT buf initial address of buffer (choice) 7 IN count number of elements in buffer (integer) 9 IN datatype datatype of each buffer element (handle) 10 OUT status status object (Status) 11 12int MPI_File_read_shared(MPI_File fh, void *buf, int count, 13 MPI_Datatype datatype, MPI_Status *status) 1415MPI_File_read_shared(fh, buf, count, datatype, status, ierror) 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 MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 2324<type> BUF(*) 25INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 26MPI_FILE_READ_SHARED reads a file using the shared file pointer. 2728 29 MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status) 30 INOUT fh file handle (handle) 31IN buf initial address of buffer (choice) 32 33 IN count number of elements in buffer (integer) 34 IN datatype datatype of each buffer element (handle) 35 OUT status status object (Status) 36 37 int MPI_File_write_shared(MPI_File fh, const void *buf, int count, 38 MPI_Datatype datatype, MPI_Status *status) 39 40 MPI_File_write_shared(fh, buf, count, datatype, status, ierror) 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 4748 MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

**Unofficial Draft for Comment Only** 

1

3

```
1
 <type> BUF(*)
\mathbf{2}
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
3
 MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
4
5
6
 MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)
7
 INOUT
 fh
 file handle (handle)
8
 OUT
 buf
9
 initial address of buffer (choice)
10
 IN
 count
 number of elements in buffer (integer)
11
 IN
 datatype
 datatype of each buffer element (handle)
12
 OUT
13
 request
 request object (handle)
14
15
 int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
16
 MPI_Datatype datatype, MPI_Request *request)
17
 MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
 INTEGER, INTENT(IN) :: count
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 TYPE(MPI_Request), INTENT(OUT) ::
 request
23
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
^{24}
25
 MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
26
 <type> BUF(*)
27
 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
28
 MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
29
 interface.
30
^{31}
32
 MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
33
 INOUT
 fh
 file handle (handle)
34
35
 IN
 buf
 initial address of buffer (choice)
36
 IN
 count
 number of elements in buffer (integer)
37
 IN
 datatype
 datatype of each buffer element (handle)
38
39
 OUT
 request
 request object (handle)
40
41
 int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
42
 MPI_Datatype datatype, MPI_Request *request)
43
 MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
44
 TYPE(MPI_File), INTENT(IN) :: fh
45
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
46
 INTEGER, INTENT(IN) :: count
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

	MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	-	$\frac{1}{2}$
INILO	Lit, OFFICIAL, INTENT(001)		3
	MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)		
• •	> BUF(*)		5
INTEG	ER FH, COUNT, DATATYPE, F	REQUEST, IERROR	6
MPI F	FILE_IWRITE_SHARED is a m	nonblocking version of the	7
	WRITE_SHARED interface.		8
	· _		9
Collective (	Operations		10
			11
		ng a shared file pointer is that the accesses to the	12
		e ranks of the processes within the group. For each	13
÷ ,		data is accessed is the position at which the shared whose ranks within the group less than that of this	14
-	_	ition, in order to prevent subsequent shared offset	15
-		erfering with this collective access, the call might	16 17
ě	-	the group have initiated their accesses. When the	18
ě	-	s to the next etype accessible, according to the file	19
	by all processes, after the last	, , <u>,</u>	20
			21
Advi	ce to users. There may be s	some programs in which all processes in the group	22
	0	shared file pointer, but the program may not <i>re</i> -	23
-		der of process rank. In such programs, using the	24
shared ordered routines (e.g., MPI_FILE_WRITE_ORDERED rather than			25
MPI_FILE_WRITE_SHARED) may enable an implementation to optimize access, im-		26	
provi	ing performance. (End of adva	ice to users.)	27
A davi	ce to implementare Accessos	to the data requested by all processes do not have	28
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			29
	-	and accesses can proceed independently from each	30
	, possibly in parallel. (End of	· · ·	31
		,	32
			$33 \\ 34$
	READ ORDERED/fh huf and	unt datatura atatua)	35
	READ_ORDERED(fh, buf, cou		36
INOUT	fh	file handle (handle)	37
OUT	buf	initial address of buffer (choice)	38
IN	count	number of elements in buffer (integer)	39
IN	datatype	datatype of each buffer element (handle)	40
OUT	status	status object (Status)	41 42
001	Status	Status Object (Status)	42
int MDT T	ile read ordered (MPI File	e fh, void *buf, int count,	43
THO RET L		e, MPI_Status *status)	45
			46
	MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) 47		
TYPE(	<pre>MPI_File), INTENT(IN) ::</pre>	fh	48

```
552
```

```
1
 TYPE(*), DIMENSION(..) :: buf
\mathbf{2}
 INTEGER, INTENT(IN) :: count
3
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
 TYPE(MPI_Status) :: status
5
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
6
 MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
7
 <type> BUF(*)
8
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
9
10
 MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED
11
 interface.
12
13
 MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
14
15
 INOUT
 fh
 file handle (handle)
16
 IN
 buf
 initial address of buffer (choice)
17
 IN
 count
 number of elements in buffer (integer)
18
19
 IN
 datatype of each buffer element (handle)
 datatype
20
 OUT
 status
 status object (Status)
21
22
 int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
23
 MPI_Datatype datatype, MPI_Status *status)
^{24}
25
 MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
26
 TYPE(MPI_File), INTENT(IN) :: fh
27
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
28
 INTEGER, INTENT(IN) :: count
29
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
 TYPE(MPI_Status) :: status
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
 MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
33
 <type> BUF(*)
34
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
36
 MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
37
 interface.
38
39
 Seek
40
 If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
41
 to call the following two routines (MPI_FILE_SEEK_SHARED and
42
 MPI_FILE_GET_POSITION_SHARED).
43
44
45
46
47
48
```

MPI_FILE_SEEK_SHARED(fh, offset, whence) ¹			
INOUT	fh	file handle (handle)	2
IN	offset	file offset (integer)	3 4
IN	whence	update mode (state)	5
	Whence		6
int MPI_F	lile_seek_shared(MPI_File	fh, MPI_Offset offset, int whence)	7
MDT File	_seek_shared(fh, offset,	whence jerror)	8 9
	(MPI_File), INTENT(IN) ::		9 10
	<pre>SER(KIND=MPI_OFFSET_KIND)</pre>		11
INTEG	ER, INTENT(IN) :: whence	e	12
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	13
MPI_FILE_	SEEK_SHARED(FH, OFFSET,	WHENCE, IERROR)	14
	ER FH, WHENCE, IERROR		15 16
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	17
MPI_F	FILE_SEEK_SHARED updates	the shared file pointer according to whence, which	18
has the fol	lowing possible values:		19
MPI	SEEK_SET: the pointer is set	to offset	20
	-		21
<ul> <li>MPI_</li> </ul>	SEEK_CUR: the pointer is set	to the current pointer position plus offset	22 23
<ul> <li>MPI_</li> </ul>	SEEK_END: the pointer is set	to the end of file plus offset	24
	-	-	25
MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values			26
for offset and whence.			27
		llows seeking backwards. It is erroneous to seek to	28
	position in the view.	5	29 30
			31
MPI FILF	_GET_POSITION_SHARED(fh	offset)	32
IN	fh	file handle (handle)	33
			34
OUT	offset	offset of shared pointer (integer)	35
	vile met menitien ebened()		36 37
int MPI_F	lie_get_position_snared()	MPI_File fh, MPI_Offset *offset)	38
	get_position_shared(fh,		39
	<pre>[MPI_File), INTENT(IN) :: ER(KIND=MPI_OFFSET_KIND)</pre>		40
	ER, OPTIONAL, INTENT(OUT)	-	41
			42 43
	GET_POSITION_SHARED(FH, )	DFFSET, IERROR)	43
	ER FH, IERROR ER(KIND=MPI_OFFSET_KIND)	OFFSET	45
			46
		D returns, in offset, the current position of the	47
shared me	pointer in etype units relative	e 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

8 MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-9 cesses using split collective data access routines. These routines are referred to as "split" 10 collective routines because a single collective operation is split in two: a begin routine and 11 an end routine. The begin routine begins the operation, much like a nonblocking data access 12(e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching 13 test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must 14 not use the buffer passed to a begin routine while the routine is outstanding; the operation 15must be completed with an end routine before it is safe to free buffers, etc. 16

¹⁷ Split collective data access operations on a file handle fh are subject to the semantic ¹⁸ 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.

According to the definitions in Section 2.4, the begin procedures are incomplete. They are also non-local procedures because they may or may not return before they are called in all MPI processes of the process group.

Advice to users. This is one of the exceptions in which incomplete procedures are non-local. (End of advice to users.)

- 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,"

1

2

3

4

5

6 7

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

• 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

<pre>MPI_File_read_all_begin(fh,</pre>	);
<pre> MPI_File_read_all(fh,);</pre>	
<pre> MPI_File_read_all_end(fh, .</pre>	);

is erroneous.

• In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

21The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and 22 MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). 23The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation  24 that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END) produces the result as defined for the equivalent collective routine (i.e., MPI_FILE_READ_ALL). 27For the purpose of consistency semantics (Section 13.6.1), a matched pair of split 28

collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access.

IN	fh	file handle (handle)	34
IN	offset	file offset (integer)	35 36
OUT	buf	initial address of buffer (choice)	36 37
IN	count	number of elements in buffer (integer)	38
IN	datatype	datatype of each buffer element (handle)	39
			40

int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype)

44MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) TYPE(MPI_File), INTENT(IN) :: fh 4546INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 47TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf 48 INTEGER, INTENT(IN) :: count

## **Unofficial Draft for Comment Only**

1

2

3

4

5

6 7

1415

16

17

18

19

20

2526

29

41

42

```
556
```

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
\mathbf{2}
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
4
 <type> BUF(*)
5
 INTEGER FH, COUNT, DATATYPE, IERROR
6
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
7
8
9
 MPI_FILE_READ_AT_ALL_END(fh, buf, status)
10
11
 IN
 fh
 file handle (handle)
12
 OUT
 buf
 initial address of buffer (choice)
13
 OUT
 status object (Status)
 status
14
15
16
 int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
17
 MPI_File_read_at_all_end(fh, buf, status, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
 TYPE(MPI_Status) :: status
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
24
 <type> BUF(*)
25
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
26
27
28
 MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
29
 INOUT
 fh
 file handle (handle)
30
 IN
 offset
 file offset (integer)
^{31}
32
 buf
 IN
 initial address of buffer (choice)
33
 IN
 count
 number of elements in buffer (integer)
34
 IN
 datatype
 datatype of each buffer element (handle)
35
36
37
 int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
38
 const void *buf, int count, MPI_Datatype datatype)
39
 MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
40
 TYPE(MPI_File), INTENT(IN) :: fh
41
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
42
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
43
 INTEGER, INTENT(IN) :: count
44
 TYPE(MPI_Datatype), INTENT(IN) ::
 datatype
45
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
46
47
 MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
48
 <type> BUF(*)
```

INTEG	ER FH, COUNT, DATATYPE, I	ERROR	1
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	2
			3 4
	_WRITE_AT_ALL_END(fh, buf	status)	5
	Ϋ́,	,	6
INOUT	fh	file handle (handle)	7
IN	buf	initial address of buffer (choice)	8
OUT	status	status object (Status)	10
int MPI_F	'ile_write_at_all_end(MPI_ MPI_Status *status)	File fh, const void *buf,	11 12
MDT File	write_at_all_end(fh, buf,	status ierror)	13 14
	<pre>[MPI_File), INTENT(IN) ::</pre>	fh	15
TYPE(	*), DIMENSION(), INTENT	(IN), ASYNCHRONOUS :: buf	16
	MPI_Status) :: status		17
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	18 19
	WRITE_AT_ALL_END(FH, BUF,	STATUS, IERROR)	20
• 1	> BUF(*) ER FH, STATUS(MPI_STATUS_	CT7E) TEDDOD	21
	ER FII, STRIUS(HFI_STRIUS_	Size), Tennon	22
			23 24
MPI_FILE	_READ_ALL_BEGIN(fh, buf, co	ount, datatype)	25
INOUT	fh	file handle (handle)	26
OUT	buf	initial address of buffer (choice)	27
IN	count	number of elements in buffer (integer)	28 29
IN	datatype	datatype of each buffer element (handle)	30
	51		31
int MPI_F	'ile_read_all_begin(MPI_Fi	le fh, void *buf, int count,	32
	MPI_Datatype datatype	e)	33 34
MPI_File_	read_all_begin(fh, buf, c	count, datatype, ierror)	35
	<pre>MPI_File), INTENT(IN) ::</pre>		36
	*), DIMENSION(), ASYNCH ER, INTENT(IN) :: count	RONOUS :: buf	37
	MPI_Datatype), INTENT(IN)	:: datatype	38 39
	ER, OPTIONAL, INTENT(OUT)	01	40
MDI FIIF	READ_ALL_BEGIN(FH, BUF, C	COUNT DATATYPE TERROR)	41
	<pre>&gt; BUF(*)</pre>		42
• 1	ER FH, COUNT, DATATYPE, I	ERROR	43 44
			44 45
			46
			47

```
1
 MPI_FILE_READ_ALL_END(fh, buf, status)
\mathbf{2}
 INOUT
 fh
 file handle (handle)
3
 OUT
 buf
 initial address of buffer (choice)
4
5
 OUT
 status object (Status)
 status
6
7
 int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
8
 MPI_File_read_all_end(fh, buf, status, ierror)
9
 TYPE(MPI_File), INTENT(IN) :: fh
10
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
11
 TYPE(MPI_Status) :: status
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
15
 <type> BUF(*)
16
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
17
18
19
 MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
20
21
 INOUT
 fh
 file handle (handle)
22
 IN
 buf
 initial address of buffer (choice)
23
 IN
 number of elements in buffer (integer)
 count
24
25
 IN
 datatype
 datatype of each buffer element (handle)
26
27
 int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
28
 MPI_Datatype datatype)
29
 MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
30
 TYPE(MPI_File), INTENT(IN) :: fh
^{31}
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
32
 INTEGER, INTENT(IN) :: count
33
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
35
36
 MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
37
 <type> BUF(*)
38
 INTEGER FH, COUNT, DATATYPE, IERROR
39
40
41
 MPI_FILE_WRITE_ALL_END(fh, buf, status)
42
 INOUT
 fh
 file handle (handle)
43
44
 IN
 buf
 initial address of buffer (choice)
45
 OUT
 status
 status object (Status)
46
47
48
```

int MPI_H	File_write_all_end(MPI_ MPI_Status *status	File fh, const void *buf,	1 2
TYPE	write_all_end(fh, buf, (MPI_File), INTENT(IN) (*) DIMENSION( ) INT		3 4 5
TYPE	(MPI_Status) :: status GER, OPTIONAL, INTENT(O		6 7
			8
	_WRITE_ALL_END(FH, BUF, >> BUF(*)	STATUS, IERROR)	9 10
INTEC	GER FH, STATUS(MPI_STAT	US_SIZE), IERROR	11 12
			13
MPI_FILE	_READ_ORDERED_BEGIN(	(fh, buf, count, datatype)	14
INOUT	fh	file handle (handle)	15 16
OUT	buf	initial address of buffer (choice)	17
IN	count	number of elements in buffer (integer)	18
IN	datatype	datatype of each buffer element (handle)	19
II.V	ullulype	datatype of each buller clement (handle)	20 21
int MPI_H	File_read_ordered_begin	(MPI_File fh, void *buf, int count,	22
	MPI_Datatype datat	ype)	23
MPI File	read ordered begin(fh.	buf, count, datatype, ierror)	24
	(MPI_File), INTENT(IN)	• -	25
TYPE	(*), DIMENSION(), ASY	NCHRONOUS :: buf	26 27
	GER, INTENT(IN) :: cou		28
	(MPI_Datatype), INTENT( SER, OPTIONAL, INTENT(O		29
	ER, UPIIONAL, INIENI(U		30
		BUF, COUNT, DATATYPE, IERROR)	31
• -	>> BUF(*) GER FH, COUNT, DATATYPE	ΤΕΡΡΟΡ	32
INIE	ER FR, COUNI, DAIAIFE	, IERRUR	34
			35
MPI_FILE	_READ_ORDERED_END(fh	, buf, status)	36
INOUT	fh	file handle (handle)	37
OUT	buf	initial address of buffer (choice)	38 39
			40
OUT	status	status object (Status)	41
int MPT F	File read ordered end(M	PI_File fh, void *buf, MPI_Status *status)	42
			43
	_read_ordered_end(fh, b		44
	(MPI_File), INTENT(IN) (*), DIMENSION(), ASY		45 46
	(MPI_Status) :: status		47
	ER, OPTIONAL, INTENT(O	UT) :: ierror	48

```
1
 MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
\mathbf{2}
 <type> BUF(*)
3
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
4
5
6
 MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
7
 INOUT
 fh
 file handle (handle)
8
 buf
9
 IN
 initial address of buffer (choice)
10
 IN
 count
 number of elements in buffer (integer)
11
 IN
 datatype
 datatype of each buffer element (handle)
12
13
 int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
14
 MPI_Datatype datatype)
15
16
 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
17
 TYPE(MPI_File), INTENT(IN) :: fh
18
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
19
 INTEGER, INTENT(IN) :: count
20
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
22
 MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
23
 <type> BUF(*)
^{24}
 INTEGER FH, COUNT, DATATYPE, IERROR
25
26
27
28
 MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
29
 INOUT
 fh
 file handle (handle)
30
 IN
 buf
 initial address of buffer (choice)
^{31}
32
 OUT
 status
 status object (Status)
33
34
 int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
35
 MPI_Status *status)
36
 MPI_File_write_ordered_end(fh, buf, status, ierror)
37
 TYPE(MPI_File), INTENT(IN) :: fh
38
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
39
 TYPE(MPI_Status) :: status
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
 MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
43
 <type> BUF(*)
44
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
45
46
47
48
```

## 13.5 File Interoperability

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) as well as the data conversion functions (Section 13.5.3).

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), 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). 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

#### Unofficial Draft for Comment Only

1 2

3

4

5

6

7

8

9

10

11

12

13

14

15

16 17

18 19

20

21 22

23

 24 

25

26

27

28

29

30

 31 

32

33

34

35

36

37

38

39

40

41

42

43

44

45

 $46 \\ 47$ 

1 2	application is responsible for specifying the same data representation as was used to create the file. ( <i>End of advice to users.</i> )
3	"native" Data in this representation is stored in a file exactly as it is in memory. The ad-
4	vantage of this data representation is stored in a me exactly as it is in memory. The ad-
5	lost in type conversions with a purely homogeneous environment. The disadvantage
6	
7	is the loss of transparent interoperability within a heterogeneous MPI environment.
8	Advice to users. This data representation should only be used in a homogeneous
9	MPI environment, or when the MPI application is capable of performing the data
10	type conversions itself. ( <i>End of advice to users.</i> )
11	type conversions itsen. ( <i>Dia of autice to users.</i> )
12	Advice to implementors. When implementing read and write operations on
13	top of MPI message-passing, the message data should be typed as MPI_BYTE
14	to ensure that the message routines do not perform any type conversions on the
15	data. (End of advice to implementors.)
16	
17	"internal" This data representation can be used for I/O operations in a homogeneous
18	or heterogeneous environment; the implementation will perform type conversions if
19	necessary. The implementation is free to store data in any format of its choice, with
20	the restriction that it will maintain constant extents for all predefined datatypes in any
20	one file. The environment in which the resulting file can be reused is implementation-
21	defined and must be documented by the implementation.
23	
24	Rationale. This data representation allows the implementation to perform $I/O$
24	efficiently in a heterogeneous environment, though with implementation-defined
26	restrictions on how the file can be reused. (End of rationale.)
27	Advice to implementors. Since "external32" is a superset of the functionality
28	provided by "internal," an implementation may choose to implement "internal"
29	as "external32." (End of advice to implementors.)
30	as externals2. ( <i>Dia of autice to implementors.</i> )
31	"external32" This data representation states that read and write operations convert all
32	data from and to the "external32" representation defined in Section 13.5.2. The data
33	conversion rules for communication also apply to these conversions (see Section $3.3.2$ ).
34	The data on the storage medium is always in this canonical representation, and the
	data in memory is always in the local process's native representation.
35	
36	This data representation has several advantages. First, all processes reading the file
37	in a heterogeneous MPI environment will automatically have the data converted to
38	their respective native representations. Second, the file can be exported from one MPI
39	environment and imported into any other MPI environment with the guarantee that
40	the second environment will be able to read all the data in the file.
41	The disadvantage of this data representation is that data precision and $I/O$ perfor-
42	mance may be lost in data type conversions.
43	
44	Advice to implementors. When implementing read and write operations on top
45	of MPI message-passing, the message data should be converted to and from the
46	"external $32$ " representation in the client, and sent as type MPI_BYTE. This will
47	avoid possible double data type conversions and the associated further loss of
48	precision and performance. (End of advice to implementors.)

#### Datatypes for File Interoperability 13.5.1

If the file data representation is other than "native," care must be taken in constructing 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), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. The routine MPI_FILE_GET_TYPE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user defined data representations. Otherwise, the etype and filetype must be constructed so that their typemap and extent are the same on any architecture. This can be achieved if they have an explicit upper bound and lower bound (defined using

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 34 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) 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.)

- 44 45
- 46
- 4748

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

1213

14

15

16

17

18

19

20

21

22

23

 24 

25

26

27

28

29

30

31

32

33

35

36

37

38

39

40

41

42

```
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)
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), MPI uses the dtype_file_extent_fn callback to calculate the extent.
21
22
 Advice to implementors. In the case of user-defined data representations, the extent
23
 of a derived datatype can be calculated by first determining the extents of the prede-
^{24}
 fined datatypes in this derived datatype using dtype_file_extent_fn (see Section 13.5.3).
25
 (End of advice to implementors.)
26
27
 13.5.2 External Data Representation: "external32"
28
29
 All MPI implementations are required to support the data representation defined in this
30
 section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required.
^{31}
 All floating point values are in big-endian IEEE format [37] of the appropriate size.
32
 Floating point values are represented by one of three IEEE formats. These are the IEEE
33
 "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage,
34
 respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16
35
 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous
36
 to the "Double" format. All integral values are in two's complement big-endian format. Big-
37
 endian means most significant byte at lowest address byte. For C _Bool, Fortran LOGICAL,
38
 and C++ bool, 0 implies false and nonzero implies true. C float _Complex, double
39
 _Complex, and long double _Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other
40
 complex types are represented by a pair of floating point format values for the real and
41
 imaginary components. Characters are in ISO 8859-1 format [38]. Wide characters (of type
42
 MPI_WCHAR) are in Unicode format [59].
43
 All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant
```

bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary
 parts at the most significant bit of each part.

According to IEEE specifications [37], the "NaN" (not a number) is system dependent. 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 in XDR (see https://www.ietf.org/rfc/rfc1832.txt). (End of advice to implementors.)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (*End of advice to implementors.*)

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The sizes 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 18.1.9, page 655.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the least 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 

$\frac{1}{2}$	Туре	Length	Optional Type	Length
3				
4	MPI_PACKED	1	MPI_INTEGER1	1
5	MPI_BYTE	1	MPI_INTEGER2	2
6	MPI_CHAR	1	MPI_INTEGER4	4
7	MPI_UNSIGNED_CHAR	1	MPI_INTEGER8	8
8	MPI_SIGNED_CHAR	1	MPI_INTEGER16	16
9	MPI_WCHAR	2		
10	MPI_SHORT	2	MPI_REAL2	2
10	MPI_UNSIGNED_SHORT	2	MPI_REAL4	4
	MPI_INT	4	MPI_REAL8	8
12	MPI_UNSIGNED	4	MPI_REAL16	16
13	 MPI_LONG	4	-	
14	MPI_UNSIGNED_LONG	4	MPI_COMPLEX4	2*2
15	MPI_LONG_LONG_INT	8	MPI_COMPLEX8	2*2
16	MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16	2*4
17				
18	MPI_FLOAT	4	MPI_COMPLEX32	2*16
19	MPI_DOUBLE	8		
20	MPI_LONG_DOUBLE	16		
21	MPI_C_BOOL	1		
22			C++ Trimon	Iongth
23	MPI_INT8_T	1	C++ Types	Length
24	MPI_INT16_T	2		
25	MPI_INT32_T	4	MPI_CXX_BOOL	1
26	MPI_INT64_T	8	MPI_CXX_FLOAT_COMPLEX	
27	MPI_UINT8_T	1	MPI_CXX_DOUBLE_COMPLEX	
28	MPI_UINT16_T	2	MPI_CXX_LONG_DOUBLE_COM	PLEX 2*16
29	MPI_UINT32_T	4		
30	MPI_UINT64_T	8		
31	MPI_AINT	8		
32	MPI_COUNT	8		
33	MPI_OFFSET	8		
	MPI_C_COMPLEX	2*4		
34	MPI_C_FLOAT_COMPLEX	2*4		
35	MPI_C_DOUBLE_COMPLEX	2*8		
36	MPI_C_LONG_DOUBLE_COMPLEX			
37		2.10		
38	MPI_CHARACTER	1		
39				
40	MPI_LOGICAL	4		
41	MPI_INTEGER	4		
42	MPI_REAL	4		
43	MPI_DOUBLE_PRECISION	8		
44	MPI_COMPLEX	2*4		
45	MPI_DOUBLE_COMPLEX	2*8		
46				
40		//		
	Table 13.2:	"external32"	sizes of predefined datatypes	
48				

```
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)
 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 conver-
 22
sion functions to convert all data items accessed between file data representation and na-
 23
tive representation. MPI_REGISTER_DATAREP is a local operation and only registers the
 24
```

data representation. Mill_REGISTER_DATAREL is a local operation and only registers the data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler (see Section 13.7). The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

```
31
Extent Callback
 32
 33
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
 34
 MPI_Aint *file_extent, void *extra_state);
 35
ABSTRACT INTERFACE
 36
 SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
 37
 ierror)
 38
 TYPE(MPI_Datatype) :: datatype
 39
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
 40
 INTEGER :: ierror
 41
 42
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
 43
 INTEGER DATATYPE, IERROR
 44
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
 45
```

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

25

26

27

28

29

```
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)
11
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
12
 TYPE(C_PTR), VALUE :: userbuf, filebuf
13
 TYPE(MPI_Datatype) :: datatype
14
 INTEGER :: count, ierror
15
 INTEGER(KIND=MPI_OFFSET_KIND) :: position
16
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
17
 SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
18
 POSITION, EXTRA_STATE, IERROR)
19
 <TYPE> USERBUF(*), FILEBUF(*)
20
 INTEGER COUNT, DATATYPE, IERROR
21
 INTEGER(KIND=MPI_OFFSET_KIND) POSITION
22
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
23
^{24}
 The function read_conversion_fn must convert from file data representation to na-
25
 tive representation. Before calling this routine, MPI allocates and fills filebuf with count
26
 contiguous data items. The type of each data item matches the corresponding entry for the
27
 predefined datatype in the type signature of datatype. The function is passed, in extra_state,
28
 the argument that was passed to the MPI_REGISTER_DATAREP call. The function must
29
 copy all count data items from filebuf to userbuf in the distribution described by datatype,
30
 converting each data item from file representation to native representation. datatype will be
^{31}
 equivalent to the datatype that the user passed to the read function. If the size of datatype
32
 is less than the size of the count data items, the conversion function must treat datatype
33
 as being contiguously tiled over the userbuf. The conversion function must begin storing
34
 converted data at the location in userbuf specified by position into the (tiled) datatype.
35
36
 Advice to users. Although the conversion functions have similarities to MPI_PACK
37
 and MPI_UNPACK, one should note the differences in the use of the arguments count
38
 and position. In the conversion functions, count is a count of data items (i.e., count
39
 of typemap entries of datatype), and position is an index into this typemap. In
40
 MPI_PACK, incount refers to the number of whole datatypes, and position is a number
41
 of bytes. (End of advice to users.)
42
43
 Advice to implementors. A converted read operation could be implemented as follows:
44
 1. Get file extent of all data items
45
46
 2. Allocate a filebuf large enough to hold all count data items
47
 3. Read data from file into filebuf
48
```

- 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, 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.

⁵ The conversion functions must be reentrant. User defined data representations are ⁶ restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion ⁷ functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI_SUCCESS, the implementation will raise an error in the class MPI_ERR_CONVERSION.

10 11 12

13

8

9

## 13.5.4 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.

¹⁹ Compatibility can be obtained when "external32" representation is used, although ²⁰ precision may be lost and the performance may be less than when "native" representation is ²¹ used. Compatibility is guaranteed using "external32" provided at least one of the following ²² conditions is met.

 $\frac{23}{24}$ 

25

26 27

28

29

30

31

32 33

34

35 36

37

38

• The data access routines directly use types enumerated in Section 13.5.2, 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 18.1.9).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 18.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

39 40 41

42

## 13.6 Consistency and Semantics

⁴³ 13.6.1 File Consistency

⁴⁴ ⁴⁵ Consistency semantics define the outcome of multiple accesses to a single file. All file ⁴⁶ accesses in MPI are relative to a specific file handle created from a collective open. MPI ⁴⁷ provides three levels of consistency: sequential consistency among all accesses using a single ⁴⁸ file handle, sequential consistency among all accesses using file handles created from a single

operations.

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) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access

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).
```

⁹ Sequential consistency is guaranteed among accesses to a single file if for any write ¹⁰ sequence  $SEQ_1$  to the file, there is no sequence  $SEQ_2$  to the file which is *concurrent* with ¹¹  $SEQ_1$ . To guarantee sequential consistency when there are write sequences,

¹² MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency
 ¹³ of the sequences.

```
See the examples in Section 13.6.11 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
 (logical)
22
23
 int MPI_File_set_atomicity(MPI_File fh, int flag)
^{24}
25
 MPI_File_set_atomicity(fh, flag, ierror)
26
 TYPE(MPI_File), INTENT(IN) ::
 fh
27
 LOGICAL, INTENT(IN) :: flag
28
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
29
 MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
30
 INTEGER FH, IERROR
^{31}
 LOGICAL FLAG
32
33
```

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

³⁵ MPI_FILE_SET_ATOMICITY on *FH*. MPI_FILE_SET_ATOMICITY is collective; all pro-³⁶ cesses in the group must pass identical values for fh and flag. If flag is true, atomic mode is ³⁷ set; if flag is false, nonatomic mode is set.

³⁸ Changing the consistency semantics for an open file only affects new data accesses. ³⁹ All completed data accesses are guaranteed to abide by the consistency semantics in effect ⁴⁰ during their execution. Nonblocking data accesses and split collective operations that have ⁴¹ not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode ⁴² 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) ¹					
IN	fh	file handle (handle)	2		
OUT	flag	true if atomic mode, false if nonatomic mode (logical)	$\frac{3}{4}$		
001	1146		4 5		
int MPI	_File_get_at	comicity(MPI_File fh, int *flag)	6		
MDT E:1	a mot atomic	ritu(fh flog iorror)	7		
<pre>MPI_File_get_atomicity(fh, flag, ierror)     TYPE(MPI_File), INTENT(IN) :: fh</pre>					
		C(OUT) :: flag	9		
		VAL, INTENT(OUT) :: ierror	10		
			11		
		CITY(FH, FLAG, IERROR)	12		
	EGER FH, IEF	IRUR	13 14		
LUG.	ICAL FLAG		14		
MPL	_FILE_GET_A	TOMICITY returns the current consistency semantics for data access	16		
operation	operations on the set of file handles created by one collective open. If flag is true, atomic				
mode is e	enabled; if flag	g is false, nonatomic mode is enabled.	18		
			19		
MPI FIL	E_SYNC(fh)		20		
			21		
INOUT	fh	file handle (handle)	22		
	()		23		
int MPI.	_File_sync(M	<pre>//PI_File fh)</pre>	24		
MPI_File	e_sync(fh, i	Lerror)	25		
TYPI	E(MPI_File),	, INTENT(IN) :: fh	26		
INT	EGER, OPTION	NAL, INTENT(OUT) :: ierror	27 28		
MPT FTL	E_SYNC(FH, I	(ERROR)	29		
	EGER FH, IER		30		
			31		
	-	_SYNC with fh causes all previous writes to fh by the calling process e storage device. If other processes have made updates to the storage	32		
		odates become visible to subsequent reads of fh by the calling process.	33		
,	-	be necessary to ensure sequential consistency in certain cases (see	34		
above).	<b>0 0</b> may	se necessary to ensure sequencial consistency in certain cases (see	35		
/	_FILE_SYNC	is a collective operation.	36 37		
The	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				
operation					
to MPI_F	ILE_SYNC is	erroneous.	39 40		
			41		
13.6.2	Random Acce	ess vs. Sequential Files	42		
MPI dist	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					
					-
	-	s not meaningful; therefore, calls to $MPI_FILE_SEEK_SHARED$ and	47		
MPI_FILI	E_GET_POSI	TION_SHARED are erroneous, and the pointer update rules specified	48		

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.

11 12 13

1

 $\mathbf{2}$ 

3 4

5 6

7

8

9

10

## 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.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

28 29

## 13.6.4 Collective File Operations

³⁰ Collective file operations are subject to the same restrictions as collective communication ³¹ operations. For a complete discussion, please refer to the semantics set forth in Section 5.14.

³² Collective file operations are collective over a duplicate of the communicator used to ³³ open the file — this duplicate communicator is implicitly specified via the file handle ar-³⁴ gument. Different processes can pass different values for other arguments of a collective ³⁵ routine unless specified otherwise.

36 37

38

## 13.6.5 Nonblocking Collective File Operations

Nonblocking collective file operations are defined only for data access routines with explicit
 offsets and individual file pointers but not with shared file pointers.

⁴¹ Nonblocking collective file operations are subject to the same restrictions as blocking ⁴² collective I/O operations. All processes belonging to the group of the communicator that ⁴³ was used to open the file must call collective I/O operations (blocking and nonblocking) ⁴⁴ in the same order. This is consistent with the ordering rules for collective operations in ⁴⁵ threaded environments. For a complete discussion, please refer to the semantics set forth ⁴⁶ in Section 5.14.

⁴⁷ Nonblocking collective I/O operations do not match with blocking collective I/O oper ⁴⁸ ations. Multiple nonblocking collective I/O operations can be outstanding on a single file

handle. High quality MPI implementations should be able to support a large number of pending nonblocking I/O operations.

All nonblocking collective I/O calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation which may progress independently of any communication, computation, or I/O. The call returns a request handle, which must be passed to a completion call. Input buffers should not be modified and output buffers should not be accessed before the completion call returns. The same progress rules described for nonblocking collective operations apply for nonblocking collective I/O operations. For a complete discussion, please refer to the semantics set forth in Section 5.12.

### 13.6.6 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

### 13.6.7 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.

### 13.6.8 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 18.2).

0	
2	MPI specifies how the data should be laid out in a virtual file structure (the view), not
3	how that file structure is to be stored on one or more disks. Specification of the physical
4	
5	file structure was avoided because it is expected that the mapping of files to disks will be
6	system specific, and any specific control over file layout would therefore restrict program
7	portability. However, there are still cases where some information may be necessary to
8	optimize file layout. This information can be provided as <i>hints</i> specified via info when a file
9	is created (see Section $13.2.8$ ).
10	13.6.10 File Size
11	IJ.U.IU THE JIZE
12	The size of a file may be increased by writing to the file after the current end of file. The size
13	may also be changed by calling MPI <i>size changing</i> routines, such as MPI_FILE_SET_SIZE.
14	A call to a size changing routine does not necessarily change the file size. For example,
15	
16	calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the
17	size.
18	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
19	high byte be the byte in that set with the largest displacement. The file size is the larger of
20	
21	• One plus the displacement of the high byte.
22	The side immediately often the side of environmenting on MDL FUE ODEN actions of
23	• The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.
24	When applying consistency semantics, calls to MPI_FILE_SET_SIZE and
25	MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations
26	
27	that access bytes at displacements between the old and new file sizes), and
28	MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the
29	file).
30	Advice to users. Any sequence of operations containing the collective routines
31	MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such,
32	sequential consistency in nonatomic mode is not guaranteed unless the conditions in
33	Section 13.6.1 are satisfied. (End of advice to users.)
34	
35	File pointer update semantics (i.e., file pointers are updated by the amount accessed)
36	are only guaranteed if file size changes are sequentially consistent.
37	are only Sudranooca is me one onengeo are coquenciany consistent.
38	Advice to users. Consider the following example. Given two operations made by
	separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and
39	
40	an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consis-
41	tency between these two operations, the file pointer may be updated by the amount
42	requested $(10 \text{ bytes})$ even if the amount accessed is zero bytes. (End of advice to
43	users.)
44	
45	13.6.11 Examples
46	····F·

46The examples in this section illustrate the application of the MPI consistency and semantics 47guarantees. These address 48

## Unofficial Draft for Comment Only

13.6.9

1

Logical vs. Physical File Layout

1 • conflicting accesses on file handles obtained from a single collective open, and  $\mathbf{2}$ • all accesses on file handles obtained from two separate collective opens. 3 4 The simplest way to achieve consistency for conflicting accesses is to obtain sequential 5consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 6 integers. If the latter, every element of b will be 5. If nonatomic mode is set, the results of 7 the read are undefined. 8 /* Process 0 */ 9 int i, a[10]; 10 int TRUE = 1; 11 12for ( i=0;i<10;i++)</pre> 13 a[i] = 5;1415MPI_File_open( MPI_COMM_WORLD, "workfile", 16MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ); 17MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ); 18 MPI_File_set_atomicity( fh0, TRUE ); 19 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status); 20/* MPI_Barrier( MPI_COMM_WORLD ); */ 2122 /* Process 1 */ 23int b[10]; 24int TRUE = 1; 25MPI_File_open( MPI_COMM_WORLD, "workfile", 26MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ); 27MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ); 28 MPI_File_set_atomicity( fh1, TRUE ); 29 /* MPI_Barrier( MPI_COMM_WORLD ); */ 30 MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status); 3132 A user may guarantee that the write on process 0 precedes the read on process 1 by imposing 33 temporal order with, for example, calls to MPI_BARRIER. 34 35 Advice to users. Routines other than MPI_BARRIER may be used to impose temporal 36 order. In the example above, process 0 could use MPI_SEND to send a 0 byte message, 37 received by process 1 using MPI_RECV. (End of advice to users.) 38 39 Alternatively, a user can impose consistency with nonatomic mode set: 40 41 /* Process 0 */ 42int i, a[10]; for ( i=0;i<10;i++)</pre> 43 44a[i] = 5;4546

```
MPI_File_open(MPI_COMM_WORLD, "workfile",
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
 48
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
```

**Unofficial Draft for Comment Only** 

```
1
 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
\mathbf{2}
 MPI_File_sync(fh0);
3
 MPI_Barrier(MPI_COMM_WORLD);
4
 MPI_File_sync(fh0);
5
 /* Process 1 */
6
 int b[10];
7
 MPI_File_open(MPI_COMM_WORLD, "workfile",
8
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
9
 MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
10
 MPI_File_sync(fh1);
11
 MPI_Barrier(MPI_COMM_WORLD);
12
 MPI_File_sync(fh1);
13
 MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
14
15
 The "sync-barrier-sync" construct is required because:
16
17
 • The barrier ensures that the write on process 0 occurs before the read on process 1.
18
 • The first sync guarantees that the data written by all processes is transferred to the
19
 storage device.
20
21
 • The second sync guarantees that all data which has been transferred to the storage
22
 device is visible to all processes. (This does not affect process 0 in this example.)
23
 The following program represents an erroneous attempt to achieve consistency by elim-
^{24}
 inating the apparently superfluous second "sync" call for each process.
25
26
 /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
27
 /* Process 0 */
28
 int i, a[10];
29
 for (i=0;i<10;i++)</pre>
30
 a[i] = 5;
^{31}
32
 MPI_File_open(MPI_COMM_WORLD, "workfile",
33
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
34
 MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
35
 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
36
 MPI_File_sync(fh0);
37
 MPI_Barrier(MPI_COMM_WORLD);
38
39
 /* Process 1 */
40
 int b[10];
^{41}
 MPI_File_open(MPI_COMM_WORLD, "workfile",
42
 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
43
 MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
44
 MPI_Barrier(MPI_COMM_WORLD);
45
 MPI_File_sync(fh1);
46
 MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
47
48
 /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
```

The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI_FILE_SYNC blocks.

Advice to users. Some implementations may choose to implement MPI_FILE_SYNC as a temporally synchronizing function. When using such an implementation, the "sync-barrier-sync" construct above can be replaced by a single "sync." The results of using such code with an implementation for which MPI_FILE_SYNC is not temporally synchronizing is undefined. (*End of advice to users.*)

## Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file "myfile." Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
 35
MPI_File_open(MPI_COMM_WORLD, "myfile",
 36
 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
 37
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INF0_NULL);
 38
/* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
 39
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
 40
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
 41
MPI_Wait(&reqs[0], &status);
 42
MPI_Wait(&reqs[1], &status);
 43
 44
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

 $\overline{7}$ 

 24 

```
1
 int a = 4, b;
\mathbf{2}
 MPI_File_open(MPI_COMM_WORLD, "myfile",
3
 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
4
 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
\mathbf{5}
 MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
6
 MPI_Wait(&reqs[0], &status);
7
 MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
8
 MPI_Wait(&reqs[1], &status);
9
 defines the same ordering as:
10
11
 int a = 4, b;
12
 MPI_File_open(MPI_COMM_WORLD, "myfile",
13
 MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
14
 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
15
 MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status);
16
 MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status);
17
18
 Since
19
 • nonconcurrent operations on a single file handle are sequentially consistent, and
20
21
 • the program fragments specify an order for the operations,
22
23
 MPI guarantees that both program fragments will read the value 4 into b. There is no need
24
 to set atomic mode for this example.
25
 Similar considerations apply to conflicting accesses of the form:
26
 MPI_File_iwrite_all(fh,...);
27
 MPI_File_iread_all(fh,...);
28
 MPI_Waitall(...);
29
30
 In addition, as mentioned in Section 13.6.5, nonblocking collective I/O operations have
^{31}
 to be called in the same order on the file handle by all processes.
32
 Similar considerations apply to conflicting accesses of the form:
33
34
 MPI_File_write_all_begin(fh,...);
 MPI_File_iread(fh,...);
35
36
 MPI_Wait(fh,...);
37
 MPI_File_write_all_end(fh,...);
38
 Recall that constraints governing consistency and semantics are not relevant to the
39
 following:
40
41
 MPI_File_write_all_begin(fh,...);
42
 MPI_File_read_all_begin(fh,...);
43
 MPI_File_read_all_end(fh,...);
^{44}
 MPI_File_write_all_end(fh,...);
45
46
 since split collective operations on the same file handle may not overlap (see Section 13.4.5).
47
48
```

## 13.7 I/O Error Handling

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.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. The default file error handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI_FILE_NULL as the fh argument to 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.

## 13.9 Examples

13.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function compute_buffer().

## Unofficial Draft for Comment Only

 24 

 31 

2		
3		
4		
5		
6		
7		
8		
9		
10		
11	MPI_ERR_FILE	Invalid file handle
12	MPI_ERR_NOT_SAME	Collective argument not identical on all
13		processes, or collective routines called in
14		a different order by different processes
15	MPI_ERR_AMODE	Error related to the <b>amode</b> passed to
16		MPI_FILE_OPEN
17	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
18		MPI_FILE_SET_VIEW
19	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
20		a file which supports sequential access only
21	MPI_ERR_NO_SUCH_FILE	File does not exist
22	MPI_ERR_FILE_EXISTS	File exists
23	MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
24	MPI_ERR_ACCESS	Permission denied
25	MPI_ERR_NO_SPACE	Not enough space
26	MPI_ERR_QUOTA	Quota exceeded
27	MPI_ERR_READ_ONLY	Read-only file or file system
28	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
29		the file is currently open by some process
30		Conversion functions could not be regis-
31	MPI_ERR_DUP_DATAREP	8
32		tered because a data representation identi-
33		fier that was already defined was passed to
34		MPI_REGISTER_DATAREP
35	MPI_ERR_CONVERSION	An error occurred in a user supplied data
36		conversion function. Other $L(Q)$ are a
37	MPI_ERR_IO	Other I/O error
38	Table 13.3	B: I/O Error Classes
39		
40		
41		
42		
43		
43		
44 45		
45		
40		
48		
-10		

```
1
2
*
* Function:
 3
 double_buffer
 4
* Synopsis:
 5
 6
 void double_buffer(
*
 7
 MPI_File fh,
 ** IN
*
 8
*
 MPI_Datatype buftype,
 ** IN
 9
 int bufcount
 ** IN
*
 10
*
)
 11
*
* Description:
 12
 Performs the steps to overlap computation with a collective write
 13
*
 by using a double-buffering technique.
 14
*
 15
*
 16
* Parameters:
 17
 fh
 previously opened MPI file handle
*
 18
 buftype
*
 MPI datatype for memory layout
 19
 (Assumes a compatible view has been set on fh)
*
*
 bufcount
 # buftype elements to transfer
 20
 21
-----/
 22
 23
/* this macro switches which buffer "x" is pointing to */
 ^{24}
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
 25
 26
void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
{
 27
 28
 29
 MPI_Status status;
 /* status for MPI calls */
 float *buffer1, *buffer2; /* buffers to hold results */
 30
 31
 float *compute_buf_ptr; /* destination buffer */
 /* for computing */
 32
 33
 float *write_buf_ptr; /* source for writing */
 34
 /* determines when to quit */
 int done;
 35
 /* buffer initialization */
 36
 37
 buffer1 = (float *)
 38
 malloc(bufcount*sizeof(float));
 39
 buffer2 = (float *)
 malloc(bufcount*sizeof(float));
 40
 41
 compute_buf_ptr = buffer1; /* initially point to buffer1 */
 42
 write_buf_ptr = buffer1; /* initially point to buffer1 */
 43
 44
 45
 /* DOUBLE-BUFFER prolog:
 46
 *
 compute buffer1; then initiate writing buffer1 to disk
 47
 */
 48
 compute_buffer(compute_buf_ptr, bufcount, &done);
```

```
1
 MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
\mathbf{2}
3
 /* DOUBLE-BUFFER steady state:
4
 *
 Overlap writing old results from buffer pointed to by write_buf_ptr
5
 *
 with computing new results into buffer pointed to by compute_buf_ptr.
6
 *
7
 *
 There is always one write-buffer and one compute-buffer in use
8
 *
 during steady state.
9
 */
10
 while (!done) {
11
 TOGGLE_PTR(compute_buf_ptr);
12
 compute_buffer(compute_buf_ptr, bufcount, &done);
13
 MPI_File_write_all_end(fh, write_buf_ptr, &status);
14
 TOGGLE_PTR(write_buf_ptr);
15
 MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
16
 }
17
18
 /* DOUBLE-BUFFER epilog:
19
 wait for final write to complete.
 *
20
 */
21
 MPI_File_write_all_end(fh, write_buf_ptr, &status);
22
23
^{24}
 /* buffer cleanup */
25
 free(buffer1);
26
 free(buffer2);
27
 }
28
29
 13.9.2 Subarray Filetype Constructor
30
^{31}
32
33
34
35
36
37
38
39
40
41
 Process 0
 Process 2
42
 Process 1
 Process 3
43
44
 Figure 13.4: Example array file layout
45
46
 Assume we are writing out a 100x100 2D array of double precision floating point num-
47
 bers that is distributed among 4 processes such that each process has a block of 25 columns
48
```

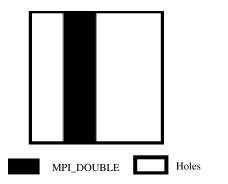


Figure 13.5: Example local array filetype for process 1

(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):

```
17
double subarray[100][25];
 18
MPI_Datatype filetype;
 19
int sizes[2], subsizes[2], starts[2];
 20
int rank;
 21
 22
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 23
sizes[0]=100; sizes[1]=100;
 ^{24}
subsizes[0]=100; subsizes[1]=25;
 25
starts[0]=0; starts[1]=rank*subsizes[1];
 26
 27
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
 28
 MPI_DOUBLE, &filetype);
 29
 30
 Or, equivalently in Fortran:
 ^{31}
 32
 double precision subarray(100,25)
 33
 integer filetype, rank, ierror
 34
 integer sizes(2), subsizes(2), starts(2)
 35
 36
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
 37
 sizes(1)=100
 sizes(2)=100
 38
 39
 subsizes(1)=100
 40
 subsizes(2)=25
 41
 starts(1)=0
 42
 starts(2)=rank*subsizes(2)
 43
 44
 call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
 MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
 45
 &
 46
 filetype, ierror)
 47
```

The generated filetype will then describe the portion of the file contained within the

## Unofficial Draft for Comment Only

9 10

11 12 13

14

15

16

1 2	process's subarray with holes for the space taken by the other processes. the filetype created for process 1.	Figure 13.5 shows
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 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 specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 18.1.5.

- 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.

Unofficial Draft for Comment Only

 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.

11 12

1

2

3

4

5

6

7

8

9

10

13 14

14.2.2 Discussion

15The objective of the MPI profiling interface is to ensure that it is relatively easy for authors 16of profiling (and other similar) tools to interface their codes to MPI implementations on 17different machines. 18

5. provide a no-op routine MPI_PCONTROL in the MPI library.

Since MPI is a machine independent standard with many different implementations, 19 it is unreasonable to expect that the authors of profiling tools for MPI will have access to 20the source code that implements MPI on any particular machine. It is therefore necessary 21to provide a mechanism by which the implementors of such tools can collect whatever 22 performance information they wish without access to the underlying implementation. 23

We believe that having such an interface is important if MPI is to be attractive to end 24users, since the availability of many different tools will be a significant factor in attracting 25users to the MPI standard. 26

The profiling interface is just that, an interface. It says *nothing* about the way in which 27it is used. There is therefore no attempt to lay down what information is collected through 28the interface, or how the collected information is saved, filtered, or displayed. 29

While the initial impetus for the development of this interface arose from the desire to 30 permit the implementation of profiling tools, it is clear that an interface like that specified  31 may also prove useful for other purposes, such as "internetworking" multiple MPI imple-32 mentations. Since all that is defined is an interface, there is no objection to its being used 33 wherever it is useful. 34

As the issues being addressed here are intimately tied up with the way in which ex-35 ecutable images are built, which may differ greatly on different machines, the examples 36 given below should be treated solely as one way of implementing the objective of the MPI 37 profiling interface. The actual requirements made of an implementation are those detailed 38 in the Requirements section above, the whole of the rest of this section is only present as 39 justification and discussion of the logic for those requirements. 40

The examples below show one way in which an implementation could be constructed to 41 meet the requirements on a Unix system (there are doubtless others that would be equally 42valid). 43

44Logic of the Design 14.2.3 45

46 Provided that an MPI implementation meets the requirements above, it is possible for 47the implementor of the profiling system to intercept the MPI calls that are made by the 48

user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

## 14.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of MPI_PCONTROL.

```
MPI_PCONTROL(level, ...)
```

IN	level			Profiling level (integer)
int	MPI_Pcontrol(const	int	level,	)

```
MPI_Pcontrol(level)
 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.
   37
   38
- 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 has been initialized 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

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

39 40

41 42

43 44

## 14.2.5 Profiler Implementation Example

A profiler can accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function as the following example shows:

```
\mathbf{5}
 Example 14.1
6
7
 static int totalBytes = 0;
 static double totalTime = 0.0;
8
9
10
 int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
11
 int dest, int tag, MPI_Comm comm)
12
 {
 /* Pass on all arguments */
13
 double tstart = MPI_Wtime();
14
 int size;
15
 = PMPI_Send(buffer,count,datatype,dest,tag,comm);
 int result
16
17
 totalTime += MPI_Wtime() - tstart;
 /* and time
 */
18
19
 MPI_Type_size(datatype, &size); /* Compute size */
20
 totalBytes += count*size;
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 preprocessor as the following example shows:

## Example 14.3

```
#ifdef PROFILELIB
ifdef __STDC__
define FUNCTION(name) P##name
else
define FUNCTION(name) P/**/name
endif
#else
define FUNCTION(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
```

/* Useful content */
}

The same source file can then be compiled to produce both versions of the library, depending on the state of the **PROFILELIB** macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

## % cc ... -lmyprof -lpmpi -lmpi

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the "name shifted" MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

## 14.2.7 Complications

## Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI func-tions (e.g., a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to "double counting" of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g., it might allow one to answer the question "How much time is spent in the point to point routines when they are called from collective functions?"), we have decided not to enforce any restrictions on the author of the MPI library that would

## 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 copied out of the base library and into the profiling one using a tool such as **ar**.

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 specific procedure names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 18.1.5.

 $\frac{34}{35}$ 

## 14.2.8 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

40 41

42

43

- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and
 it is not the business of a standard to require a particular implementation language, we
 decided to accept the scheme outlined above.

⁴⁷ Note, however, that it is possible to use the scheme above to implement a multi-level
 ⁴⁸ system, since the function called by the user may call many different profiling functions

before calling the underlying MPI function. This capability has been demonstrated in the  $P^N$ MPI tool infrastructure [51].

## 14.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance. Understanding and manipulating these variables can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation. The interface is split into two parts: the first part provides information about and supports the setting of control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control and performance variables exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables that exist in a particular MPI implementation, to query their properties, to retrieve descriptions about their meaning, and to access and, if appropriate, to alter their values.

Variables and categories across connected MPI processes with equivalent names are required to have the same meaning (see the definition of "equivalent" as related to strings in Section 14.3.3). Furthermore, enumerations with equivalent names across connected MPI processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected MPI processes in enumerations with the same meaning must also have the same meaning. In order for variables and categories to have the same meaning, routines in the tools information interface that return details for those variables and categories have requirements on what parameters must be identical. These requirements are specified in their respective sections.

*Rationale.* The intent of requiring the same meaning for entities with equivalent names is to enforce consistency across connected MPI processes. For example, variables describing the number of packets sent on different types of network devices should have different names to reflect their potentially different meanings. (*End of rationale.*)

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 is initialized and after MPI is finalized. 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. 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

¹ Since the MPI tool information interface primarily focuses on tools and support li-² braries, MPI implementations are only required to provide C bindings for functions and ³ constants introduced in this section. Except where otherwise noted, all conventions and ⁴ principles governing the C bindings of the MPI API also apply to the MPI tool information ⁵ interface, which is available by including the mpi.h header file. All routines in this interface ⁶ 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 the number of variables and variable indices are the same across connected MPI processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable. (*End of advice to users.*)

14.3.1 Verbosity Levels

22 The MPI tool information interface provides access to internal configuration and performance information through a set of control and performance variables defined by the MPI 23 24 implementation. Since some implementations may export a large number of variables, 25variables are classified by a verbosity level that categorizes both their intended audience 26(end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. 27Table 14.1 lists the constants for all possible verbosity levels. The values of the con-28stants are monotonic in the order listed in the table; i.e., MPI_T_VERBOSITY_USER_BASIC 29 < MPI_T_VERBOSITY_USER_DETAIL < ... < MPI_T_VERBOSITY_MPIDEV_ALL. 30

31		
32 MPI_T	_VERBOSITY_USER_BASIC	Basic information of interest to users
33 MPI_T	_VERBOSITY_USER_DETAIL	Detailed information of interest to users
34 MPI_T	_VERBOSITY_USER_ALL	All remaining information of interest to users
35 MPI_T	_VERBOSITY_TUNER_BASIC	Basic information required for tuning
36 MPI_T	_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
37 MPI_T	_VERBOSITY_TUNER_ALL	All remaining information required for tuning
38 MPI_T	_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
39 MPI_T	_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
40 MPI_T	_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors

Table 14.1: MPI tool information interface verbosity levels

## 14.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

Each MPI tool information interface variable provides access to a particular control setting
 or performance property of the MPI implementation. A variable may refer to a specific

7

8

9

10

11

12

13 14

15

16

17

18

19 20

21

41 42

43 44 45

MPI object such as a communicator, datatype, or one-sided communication window, or the variable may refer more generally to the MPI environment of the process. Except for the last case, the variable must be bound to exactly one MPI object before it can be used. Table 14.2 lists all MPI object types to which an MPI tool information interface variable can be bound, together with the matching constant that MPI tool information interface routines return to identify the object type.

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

*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 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.

## Unofficial Draft for Comment Only

 24 

¹ MPI implementations behave as if they have an internal character array that is copied ² to the output character array supplied by the user. Such output strings are only defined ³ to be equivalent if their notional source-internal character arrays are identical (up to and ⁴ including the null terminator), even if the output string is truncated due to a small input ⁵ length parameter n.

14.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization routines.

10 11 12

13 14 15

6 7

8

9

MPI_T_INIT_THREAD(required, provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

16 17

18

37

38 39

40

41

42

43

int MPI_T_init_thread(int required, int *provided)

All programs or tools that use the MPI tool information interface must initialize the 19MPI tool information interface in the processes that will use the interface before calling 20any other of its routines. A user can initialize the MPI tool information interface by calling 21MPI_T_INIT_THREAD, which can be called multiple times. In addition, this routine initial-22izes the thread environment for all routines in the MPI tool information interface. Calling 23this routine when the MPI tool information interface is already initialized has no effect  24 beyond increasing the reference count of how often the interface has been initialized. The 25argument required is used to specify the desired level of thread support. The possible values 26and their semantics are identical to the ones that can be used with MPI_INIT_THREAD 27listed in Section 12.4. The call returns in provided information about the actual level of 28thread support that will be provided by the MPI implementation for calls to MPI tool 29information interface routines. It can be one of the four values listed in Section 12.4. 30

The MPI specification does not require all MPI processes to exist before MPI is initialized. If the MPI tool information interface is used before initialization of MPI, 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 initialization 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 initialization (instead of adding them
 during initialization) to allow tools the most flexibility. In particular, control variables
 should be available before MPI initialization if their value cannot be changed after MPI
 initialization. (End of advice to implementors.)

#### 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 interface is no longer initialized. The user can reinitialize the interface by a subsequent call 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 initialization. Consequently, these routines can also use MPI datatypes before MPI initialization. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before MPI initialization.

MPI_INT	
MPI_UNSIGNED	
MPI_UNSIGNED_LONG	
MPI_UNSIGNED_LONG_LONG	
MPI_COUNT	
MPI_CHAR	

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.*)

 $\overline{7}$ 

 24 

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.

4 The use of the datatype MPI_CHAR in the MPI tool information interface implies a null- $\mathbf{5}$ terminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR, 6 the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and 7MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including

8 its terminating null character. The contents of returned MPI_CHAR arrays are only defined 9 from index 0 through the location of the first null character. 10

The MPI tool information interface requires a significantly simpler type Rationale. system than MPI itself. Therefore, only its required subset must be present before MPI initialization 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 infor-16mation by associating names with a fixed number of values. We refer to this information 17in the following as an enumeration. In this case, the respective calls that provide addi-18 tional metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO 19 (Section 14.3.6) and MPI_T_PVAR_GET_INFO (Section 14.3.7), return a handle of type 20MPI_T_enum that can be passed to the following functions to extract additional informa-21tion. Thus, the MPI implementation can describe variables with a fixed set of values that 22 each represents a particular state. Each enumeration type can have N different values, with 23a fixed N that can be queried using  $MPI_T_ENUM_GET_INFO$ .  24 

2526

27

36

37

38

39

40

41

42

MPI_T_ENUM_GET_INFO(enumtype, num, name, name_len)

28	IN	enumtype	enumeration to be queried (handle)
29 30 31	OUT	num	number of discrete values represented by this enumer- ation (integer)
31 32 33	OUT	name	buffer to return the string containing the name of the enumeration (string)
34 35	INOUT	name_len	length of the string and/or buffer for $name$ (integer)

## int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name, int *name_len)

If enumtype is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments name and name_len are used to return the name of the enumeration as described in Section 14.3.3.

43 The routine is required to return a name of at least length one. This name must be 44unique with respect to all other names for enumerations that the MPI implementation uses. 45Names associated with individual values in each enumeration enumtype can be queried 46 using MPI_T_ENUM_GET_ITEM.

48

47

1

 $\mathbf{2}$ 

3

11

12

13

-		• • • • • • • • • • • • • • • • • • • •		
	IN	enumtype	enumeration to be queried (handle)	- 4
	IN	index	number of the value to be queried in this enumeration (integer)	4
	OUT	value	variable value (integer)	6
	OUT	name	buffer to return the string containing the name of the enumeration item (string)	7 8 9
	INOUT	name_len	length of the string and/or buffer for $name$ (integer)	1

#### MPI_T_ENUM_GET_ITEM(enumtype, index, value, name, name_len)

## 

The arguments name and name_len are used to return the name of the enumeration item as described in Section 14.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

## 14.3.6 Control Variables

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an "eager limit," i.e., an upper bound on the size of messages sent or received using an eager protocol.

## Control Variable Query Functions

An MPI implementation exports a set of N control variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any control variables, otherwise the provided control variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (*End of advice to users.*)

 $47 \\ 48$ 

## Unofficial Draft for Comment Only

 24 

1 2	The following function can be used to query the number of control variables, $num_cvar$		
$\frac{3}{4}$	MPI_T_C\	/AR_GET_NUM(num_d	cvar)
5	OUT	num_cvar	returns number of control variables (integer)
7 8	int MPI_7	_cvar_get_num(int	*num_cvar)
9 10 11	The f each varial		$_GET_INFO$ provides access to additional information for
12 13 14	MPI_T_C\	/AR_GET_INFO(cvar_i desc_len, bind, sc	ndex, name, name_len, verbosity, datatype, enumtype, desc, cope)
15 16	IN	cvar_index	index of the control variable to be queried, value be- tween 0 and $num_cvar - 1$ (integer)
17 18	OUT	name	buffer to return the string containing the name of the control variable (string)
19 20	INOUT	name_len	length of the string and/or buffer for name (integer)
20 21	OUT	verbosity	verbosity level of this variable (integer)
22 23	OUT	datatype	MPI data type of the information stored in the control variable (handle)
24 25	OUT	enumtype	optional descriptor for enumeration information (han- dle)
26 27 28	OUT	desc	buffer to return the string containing a description of the control variable (string)
29	INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)
30 31	OUT	bind	type of MPI object to which this variable must be bound (integer)
32 33 34	OUT	scope	scope of when changes to this variable are possible (integer)
35 36 37 38	int MPI_7	int *verbosity	<pre>cvar_index, char *name, int *name_len, , MPI_Datatype *datatype, MPI_T_enum *enumtype, t *desc_len, int *bind, int *scope)</pre>
39			thm: thm: thm: thm: thm: thm: thm: thm:
40 41		* 0	nformation about the same variable must return the same
41		=	ation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implemen-
43			and not return a value for the parameter.
44			me_len are used to return the name of the control variable
45		ed in Section $14.3.3$ .	
46 47			ne routine is required to return a name of at least length
47 48		name must be unique I implementation.	with respect to all other names for control variables used

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1).

The argument datatype returns the MPI datatype that is used to represent the control variable.

If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 14.3.5. Otherwise, enumtype is set to MPI_T_ENUM_NULL. If the datatype is not MPI_INT or the argument enumtype is the null pointer, no enumeration type is returned.

The arguments desc and desc_len are used to return a description of the control variable as described in Section 14.3.3.

Returning a description is optional. If an MPI implementation does not 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 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 MPI process or must be done by the user across multiple connected MPI processes. The latter is further split into variables that require changes in a group of MPI processes and those that require collective changes among all connected MPI processes. Both cases can require variables on all participating MPI processes either to be set to consistent (but potentially different) values or to equal values. 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.

If the name of a control variable is equivalent across connected MPI processes, the following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope. The returned description must be equivalent.

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 set to consistent values
	across a group of connected MPI processes
MPI_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value
	across a group of connected MPI processes
MPI_T_SCOPE_ALL	may be writeable, must be set to consistent values
	across all connected MPI processes
MPI_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value
	across all connected MPI processes

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.*)

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

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 24 

1 MPI_T_CVAR_GET_INDEX(name, cvar_index) 2 IN name of the control variable (string) name 3 OUT cvar_index index of the control variable (integer) 4 56 int MPI_T_cvar_get_index(const char *name, int *cvar_index) 7 MPI_T_CVAR_GET_INDEX is a function for retrieving the index of a control variable 8 given a known variable name. The name parameter is provided by the caller, and cvar_index 9 is returned by the MPI implementation. The name parameter is a string terminated with a 10 null character. 11 This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME 12if name does not match the name of any control variable provided by the implementation 13 at the time of the call. 14 15Rationale. This routine is provided to enable fast retrieval of control variables by 16a tool, assuming it knows the name of the variable for which it is looking. The 17 number of variables exposed by the implementation can change over time, so it is not 18 possible for the tool to simply iterate over the list of variables once at initialization. 19 Although using MPI implementation specific variable names is not portable across MPI 20implementations, tool developers may choose to take this route for lower overhead at 21runtime because the tool will not have to iterate over the entire set of variables to 22 find a specific one. (End of rationale.) 23 24 Example: Printing All Control Variables 2526Example 14.4 27The following example shows how the MPI tool information interface can be used to 28query and to print the names of all available control variables. 29 30 #include <stdio.h> 31 #include <stdlib.h> 32 #include <mpi.h> 33 34 int main(int argc, char *argv[]) { 35 int i, err, num, namelen, bind, verbose, scope; 36 int threadsupport; 37 char name[100]; 38 MPI_Datatype datatype; 39 40err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport); 41 if (err!=MPI_SUCCESS) 42return err; 43 44 err=MPI_T_cvar_get_num(&num); 45 if (err!=MPI_SUCCESS) 46 return err; 47

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user must first allocate a handle of type MPI_T_cvar_handle for the variable by binding it to an MPI object (see also Section 14.3.2).

*Rationale.* Handles used in the MPI tool information interface are distinct from handles used in the remaining parts of the MPI standard because they must be usable before MPI is initialized and after MPI is finalized. Further, accessing handles, in particular for performance variables, can be time critical and having a separate handle space enables optimizations. (*End of rationale.*)

MPI_T_CVAR_HANDLE_ALLOC(cvar_	index, ob	oj_handle,	handle,	count)
-------------------------------	-----------	------------	---------	--------

	( -		30
IN	cvar_index	index of control variable for which handle is to be al-	31
		located (index)	32
IN	obj_handle	reference to a handle of the MPI object to which this	33
		variable is supposed to be bound (pointer)	34
OUT	handle	allocated handle (handle)	35
OUT	count	number of elements used to represent this variable (in-	36
001	count		37
		teger)	38
			39

This routine binds the control variable specified by the argument index to an MPI object. ⁴² The object is passed in the argument obj_handle as an address to a local variable that stores ⁴³ the object's handle. The argument obj_handle is ignored if the MPI_T_CVAR_GET_INFO ⁴⁴ call for this control variable returned MPI_T_BIND_NO_OBJECT in the argument bind. The ⁴⁵ 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 ⁴⁷ MPI_T_CVAR_GET_INFO call) used to represent this variable. ⁴⁸

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

1 2 3	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.				
4 5 6 7 8 9	It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD to this routine, since their implementation depends on the MPI library. Instead, such object handles should be stored in a local variable and the address of this local variable should be passed into MPI_T_CVAR_HANDLE_ALLOC. ( <i>End of advice to users.</i> )				
10 11 12 13 14	is the num MPI_T_CV	ber of available AR_GET_NUM.	c should be in the range 0 to $num_cvar - 1$ , where $num_cv$ control variables as determined from a prior call to The type of the MPI object it references must be consistence the bind argument in a prior call to MPI_T_CVAR_GET_INFO	ent	
15 16	MPI_T_C\	/AR_HANDLE_F	REE(handle)		
17 18	INOUT	handle	handle to be freed (handle)		
19	int MPI_T	_cvar_handle_	ree(MPI_T_cvar_handle *handle)		
20 21 22 23 24	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.				
25 26 27	Control Va	riable Access Fur	tions		
28 29	MPI_T_C\	/AR_READ(hand	e, buf)		
30	IN	handle	handle to the control variable to be read (handle)		
31 32 33	OUT	buf	initial address of storage location for variable val (choice)	lue	
34 35	int MPI_T	_cvar_read(MP	_T_cvar_handle handle, void* buf)		
36 37 38 39 40 41 42	This routine queries the value of a control variable identified by the argument handle and stores the result 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 control variable (based on the returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO and MPI_T_CVAR_HANDLE_ALLOC, respectively).				
43	MPI_T_C\	/AR_WRITE(han	le, buf)		
44 45	IN	handle	handle to the control variable to be written (handle	e)	
46 47 48	IN	buf	initial address of storage location for variable val (choice)	lue	

#### int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf)

This routine sets the value of the control variable identified by the argument handle to the data stored 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 control variable (based on the returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO and MPI_T_CVAR_HANDLE_ALLOC, respectively).

If the variable has a global scope (as returned by a prior corresponding MPI_T_CVAR_GET_INFO call), any write call to this variable must be issued by the user in all connected (as defined in Section 10.5.4) MPI processes. If the variable has group scope, any write call to this variable must be issued by the user in all MPI processes in the group, which must be described by the MPI implementation in the description by the MPI_T_CVAR_GET_INFO.

In both cases, the user must ensure that the writes in all participating MPI processes are consistent. If the scope is either MPI_T_SCOPE_ALL_EQ or MPI_T_SCOPE_GROUP_EQ this means that the variable in all connected MPI processes or MPI processes of the group, respectively, must be set to the same value.

If it is not possible to change the variable at the time the call is made, the function returns either MPI_T_ERR_CVAR_SET_NOT_NOW, if there may be a later time at which the variable could be set, or MPI_T_ERR_CVAR_SET_NEVER, if the variable cannot be set for the remainder of the application's execution.

Example: Reading the Value of a Control Variable

#### Example 14.5

}

The following example shows a routine that can be used to query the value with a control variable with a given index. The example assumes that the variable is intended to be bound to an MPI communicator.

```
int getValue_int_comm(int index, MPI_Comm comm, int *val) {
 int err,count;
 MPI_T_cvar_handle handle;
```

```
/* This example assumes that the variable index */
/* can be bound to a communicator */
```

```
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;
```

 $\mathbf{2}$ 

#### 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.)

13 Performance Variable Classes 14

15Each performance variable is associated with a class that describes its basic semantics, 16possible datatypes, basic behavior, its starting value, whether it can overflow, and when 17and how an MPI implementation can change the variable's value. The starting value is the 18 value that is assigned to the variable the first time that it is used or whenever it is reset.

1920

2122

23

24

25

2627

2829

30

31

32

33

34

35

36 37

38

39

40

41

42

43

44

45

46

48

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (End of advice to users.)

MPI implementations should use large enough datatypes Advice to implementors. for each performance variable to avoid overflows under normal circumstances. (End of advice to implementors.)

The classes are defined by the following constants:

## MPI_T_PVAR_CLASS_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 14.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI_T_PVAR_CLASS_LEVEL

A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI T PVAR CLASS SIZE

47 A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are non-negative and represented by one

## **Unofficial Draft for Comment Only**

1

2

3

4

5

6

7 8

9

10

11

of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current size 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_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.

## • 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 variable is started or reset. 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 non-negative and decreases monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the variable is started or reset. 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 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,

## Unofficial Draft for Comment Only

8 9 10

11

12

13 14

15

16

17

18

19

20

23

 24 

25 26

27 28

1 2

3

4

5

6

 $\overline{7}$ 

21 22

29 30 31

32

33

34

35

36

37

38 39

> 40 41

42

43

44

45

46

47

1

2

3

4

5

6

7

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.

17 18

19

20

21

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.

#### ²² ₂₃ Performance Variable Query Functions

²⁴ An MPI implementation exports a set of N performance variables through the MPI tool ²⁵ information interface. If N is zero, then the MPI implementation does not export any ²⁶ performance variables; otherwise the provided performance variables are indexed from 0 to ²⁷ N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

- The following function can be used to query the number of performance variables, N:
- 34 35 36

37 38

39 40

41

42

43

MPI_T_PVAR_GET_NUM(num_pvar)

OUT num_pvar returns number of performance variables (integer)

```
int MPI_T_pvar_get_num(int *num_pvar)
```

The function  $\mathsf{MPI_T_PVAR_GET_INFO}$  provides access to additional information for each variable.

- 44 45
- 46
- 10
- 47 48

MPI_T_PVAR_GET_INFO(pvar_index, name, name_len, verbosity, varclass, datatype, enumtype, desc, desc_len, bind, readonly, continuous, atomic) ²				
IN	pvar_index	index of the performance variable to be queried be- tween 0 and $num_pvar - 1$ (integer)	3 4 5	
OUT	name	buffer to return the string containing the name of the performance variable (string)	6 7	
INOUT	name_len	length of the string and/or buffer for name (integer) $% \left( {{\left[ {{{\left[ {{\left[ {{\left[ {{\left[ {{\left[ {{\left[ $	8	
OUT	verbosity	verbosity level of this variable (integer)	9 10	
OUT	var_class	class of performance variable (integer)	11	
OUT	datatype	MPI data type of the information stored in the performance variable (handle)	12 13	
OUT	enumtype	optional descriptor for enumeration information (han- dle)	14 15 16	
OUT	desc	buffer to return the string containing a description of the performance variable (string)	17 18	
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)	19	
OUT	bind	type of MPI object to which this variable must be bound (integer)	20 21 22	
OUT	readonly	flag indicating whether the variable can be written/reset (integer)	23 24	
OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)	25 26	
OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)	27 28 29	

## 

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.

If any OUT parameter to MPI_T_PVAR_GET_INFO is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments name and name_len are used to return the name of the performance variable as described in Section 14.3.3. If completed successfully, the routine is required to return a name of at least length one.

The 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 perfor mance variable.

⁸ If the variable is of type MPI_INT, MPI can optionally specify an enumeration for the ⁹ values represented by this variable and return it in enumtype. In this case, MPI returns an ¹⁰ enumeration identifier, which can then be used to gather more information as described in ¹¹ Section 14.3.5. Otherwise, enumtype is set to MPI_T_ENUM_NULL. If the datatype is not ¹² MPI_INT or the argument enumtype is the null pointer, no enumeration type is returned.

Returning a description is optional. If an MPI implementation does not return a de scription, 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.

¹⁶ The parameter bind returns the type of the MPI object to which the variable must be ¹⁷ bound or the value MPI_T_BIND_NO_OBJECT (see Section 14.3.2).

¹⁸ Upon return, the argument readonly is set to zero if the variable can be written or reset ¹⁹ 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.

²⁴ Upon return, the argument **atomic** is set to zero if the variable cannot be read and ²⁵ reset atomically. Only variables for which the call sets **atomic** to one can be used in a call ²⁶ to MPI_T_PVAR_READRESET.

If a performance variable has an equivalent name and has the same class across con nected MPI processes, the following OUT parameters must be identical: verbosity, varclass,
 datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must
 be equivalent.

31 32

MPI_T_PVAR_GET_INDEX(name, var_class, pvar_index)

IN	name	the name of the performance variable (string)
IN	var_class	the class of the performance variable (integer)
OUT	pvar_index	the index of the performance variable (integer)

37 38 39

40

41

42

43

int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)

MPI_T_PVAR_GET_INDEX is a function for retrieving the index of a performance variable given a known variable name and class. The name and var_class parameters are provided by the caller, and pvar_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME if name does not match the name of any performance variable of the specified var_class provided by the implementation at the time of the call.

48

1

2

3

4

*Rationale.* This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (*End of rationale.*)

#### 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.

	<pre>/IPI_T_PVAR_SESSION_CREATE(session)</pre>				
			19		
OUT	session	identifier of performance session (handle)	20		
			21		
int MPI_	T_pvar_session_	create(MPI_T_pvar_session *session)	22		
This	call croates a new s	session for accessing performance variables and returns a handle	23		
		nent session of type MPI_T_pvar_session.	24		
for this se	ssion in the argun	tent session of type wifi_i _pvar_session.	25		
			26		
MPI T P	VAR_SESSION_FR	(EE(session)	27		
			28		
INOUT	OUT session	identifier of performance experiment session (handle)	29		
			30		
int MPI_	T_pvar_session_:	free(MPI_T_pvar_session *session)	31		
This	call frees an existi	ng session. Calls to the MPI tool information interface can no	32		
		ontext of a session after it is freed. On a successful return, MPI	33		
0	sets the session identifier to MPI_T_PVAR_SESSION_NULL.				
Sets the st			35		
Handlo All	location and Deallo	cation	36		
Tianule All	location and Deallo		37		
Before usi	ing a performance	variable, a user must first allocate a handle of type	38		
MPI_T_p	var_handle for the v	variable by binding it to an MPI object (see also Section 14.3.2).	39		
		40			
			41		
			42		
			43		

 $\mathbf{5}$ 

1	MPI_T_PV	AR_HANDLE_ALLOC(session,	pvar_index, obj_handle, handle, count)		
2	IN	session	identifier of performance experiment session (handle)		
3 4 5	IN	pvar_index	index of performance variable for which handle is to be allocated (integer)		
6 7	IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)		
8 9	OUT	handle	allocated handle (handle)		
10 11	OUT	count	number of elements used to represent this variable (in- teger)		
12 13 14	int MPI_T	-	_pvar_session session, int pvar_index, I_T_pvar_handle *handle, int *count)		
15 16 17 18 19 20 21 22 23	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.				
24 25 26	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.				
27 28 29 30 31 32	It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI_T_PVAR_HANDLE_ALLOC. ( <i>End of advice to users.</i> )				
33 34 35 36 37 38 39 40 41	The value of index should be in the range 0 to num_pvar - 1, where num_pvar is the number of available performance variables as determined from a prior call to MPI_T_PVAR_GET_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_PVAR_GET_INFO. For all routines in the rest of this section that take both handle and session as IN or INOUT arguments, if the handle argument passed in is not associated with the session argument, MPI_T_ERR_INVALID_HANDLE is returned.				
42	MPI_T_PV	AR_HANDLE_FREE(session, h	nandle)		
43	IN	session	identifier of performance experiment session (handle)		
44 45	INOUT	handle	handle to be freed (handle)		
46 47 48	<pre>int MPI_T_pvar_handle_free(MPI_T_pvar_session session,</pre>				

## 14.3. THE MPI TOOL INFORMATION INTERFACE

When a handle is no longer needed, a user of the MPI tool information interface should call MPI_T_PVAR_HANDLE_FREE to free the handle in the session identified by the parameter session and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to MPI_T_PVAR_HANDLE_NULL.

#### Starting and Stopping of Performance Variables

Performance variables that have the continuous flag set during the query operation are continuously operating once a handle has been allocated. Such variables may be queried at any time, but they cannot be started or stopped by the user. All other variables are in a stopped state after their handle has been allocated; their values are not updated until they have been started by the user.

#### MPI_T_PVAR_START(session, handle)

IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)

#### int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)

This functions starts the performance variable with the handle identified by the parameter 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 start all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are started successfully (even if there are no non-continuous variables to be started), otherwise MPI_T_ERR_PVAR_NO_STARTSTOP is returned. Continuous variables and variables that are already started are ignored when MPI_T_PVAR_ALL_HANDLES is specified.

MPI_T_PVAR_STOP(session, handle)

IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)

#### int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)

This functions stops the performance variable with the handle identified by the parameter 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 (even if there are no non-continuous variables to be stopped), 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.

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12 13 14

20

21

22

23

 24 

25

26

27

28 29 30

36

37

38

39

40

41

42

43

⁴⁵ 46

	614		CHAPTER 14. TOOL SUPPORT	
1 2 3	Performar	nce Variable Acce	ss Functions	
4	MPI_T_F	VAR_READ(sess	ion, handle, buf)	
5 6	IN	session	identifier of performance experiment session (handle)	
7	IN	handle	handle of a performance variable (handle)	
8 9 10	OUT	buf	initial address of storage location for variable value (choice)	
11 12 13	int MPI_	T_pvar_read(M_ void* bu	PI_T_pvar_session session, MPI_T_pvar_handle handle, f)	
14 15 16 17 18 19 20 21 22 23	handle ha buffer ide is of the the datat MPI_T_P The	andle in the sessi- entified by the p appropriate size type and count PVAR_GET_INFC	<b>READ</b> call queries the value of the performance variable with the on identified by the parameter <b>session</b> and stores the result in the arameter <b>buf</b> . The user is responsible to ensure that the buffer to hold the entire value of the performance variable (based on returned by the corresponding previous calls to 0 and MPI_T_PVAR_HANDLE_ALLOC, respectively). PVAR_ALL_HANDLES cannot be used as an argument for the func- b.	
24	MPI_T_P	VAR_WRITE(see	ssion,handle, buf)	
25 26	IN	session	identifier of performance experiment session (handle)	
27	IN	handle	handle of a performance variable (handle)	
28 29	IN	buf	initial address of storage location for variable value (choice)	
30 31 32	int MPI_	T_pvar_write() const vo	<pre>MPI_T_pvar_session session, MPI_T_pvar_handle handle, id* buf)</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>	with the l session. T user must formance calls to M If it MPI_T_EF The	handle identified The value to be we t ensure that the variable (based of MPI_T_PVAR_GE is not possible RR_PVAR_NO_WE	PVAR_ALL_HANDLES cannot be used as an argument for the func-	

#### 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 (even if there are no valid handles or all are read-only), 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)
IN	handle	handle of a performance variable (handle)
OUT	buf	initial address of storage location for variable value
		(choice)

#### 

This call atomically combines the functionality of MPI_T_PVAR_READ and MPI_T_PVAR_RESET with the same semantics as if these two calls were called separately. If atomic operations on this variable are not supported, this routine returns MPI_T_ERR_PVAR_NO_ATOMIC.

The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_READRESET.

Advice to implementors. Sampling-based tools rely on the ability to call the MPI tool information interface, in particular routines to start, stop, read, write, and reset performance variables, from any program context, including asynchronous contexts such 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

#### Unofficial Draft for Comment Only

 $46 \\ 47$ 

1

 $\mathbf{2}$ 

3 4

5 6

7

8

9

10

11

12

of MPI_T_PVAR_ALL_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

#### Example: Tool to Detect Receives with Long Unexpected Message Queues

#### Example 14.6

The following example shows a sample tool to identify receive operations that occur during times with long message queues. This examples assumes that the MPI implementation exports a variable with the name "MPI_T_UMQ_LENGTH" to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to MPI_INIT), (2) the test for long unexpected message queues (by intercepting calls to MPI_RECV), and (3) the clean-up phase (by intercepting the call to MPI_FINALIZE). To capture all receives, the example would have to be extended to have similar wrappers for

 $^{16}_{17}$  all receive operations.

Part 1 — Initialization: During initialization, the tool searches for the variable and, once
 the right index is found, allocates a session and a handle for the variable with the found
 index, and starts the performance variable.

```
22
 #include <stdio.h>
23
 #include <stdlib.h>
24
 #include <string.h>
25
 #include <assert.h>
26
 #include <mpi.h>
27
28
 /* Global variables for the tool */
29
 static MPI_T_pvar_session session;
30
 static MPI_T_pvar_handle handle;
31
32
 int MPI_Init(int *argc, char ***argv) {
33
 int err, num, i, index, namelen, verbosity;
34
 int var_class, bind, threadsup;
35
 int readonly, continuous, atomic, count;
36
 char name[18];
37
 MPI_Comm comm;
38
 MPI_Datatype datatype;
39
 MPI_T_enum enumtype;
40
41
 err=PMPI_Init(argc,argv);
42
 if (err!=MPI_SUCCESS) return err;
43
44
 err=PMPI_T_init_thread(MPI_THREAD_SINGLE,&threadsup);
45
 if (err!=MPI_SUCCESS) return err;
46
47
 err=PMPI_T_pvar_get_num(&num);
48
```

```
1
 if (err!=MPI_SUCCESS) return err;
 2
 index=-1;
 3
 i=0;
 while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
 4
 /* Pass a buffer that is at least one character longer than */
 5
 6
 /* the name of the variable being searched for to avoid */
 7
 /* finding variables that have a name that has a prefix */
 8
 /* equal to the name of the variable being searched. */
 9
 namelen=18;
 10
 err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
 11
 &var_class, &datatype, &enumtype, NULL, NULL, &bind,
 &readonly, &continuous, &atomic);
 12
 if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
 13
 14
 i++; }
 15
 if (err!=MPI_SUCCESS) return err;
 16
 17
 /* this could be handled in a more flexible way for a generic tool */
 18
 assert(index>=0);
 19
 assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
 assert(datatype==MPI_INT);
 20
 21
 assert(bind==MPI_T_BIND_MPI_COMM);
 22
 23
 /* Create a session */
 24
 err=PMPI_T_pvar_session_create(&session);
 25
 if (err!=MPI_SUCCESS) return err;
 26
 /* Get a handle and bind to MPI_COMM_WORLD */
 27
 comm=MPI_COMM_WORLD;
 28
 29
 err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
 30
 if (err!=MPI_SUCCESS) return err;
 31
 /* this could be handled in a more flexible way for a generic tool */
 32
 33
 assert(count==1);
 34
 /* Start variable */
 35
 err=PMPI_T_pvar_start(session, handle);
 36
 37
 if (err!=MPI_SUCCESS) return err;
 38
 39
 return MPI_SUCCESS;
 40
 41
 42
Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the
 43
```

tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

```
#define THRESHOLD 5
```

}

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,

44

45

```
1
 int tag, MPI_Comm comm, MPI_Status *status)
2
 {
3
 int value, err;
4
5
 if (comm==MPI_COMM_WORLD) {
6
 err=PMPI_T_pvar_read(session, handle, &value);
7
 if ((err==MPI_SUCCESS) && (value>THRESHOLD))
8
 {
9
 /* tool identified receive called with long UMQ */
10
 /* execute tool functionality, */
11
 /* e.g., gather and print call stack */
12
 }
 }
13
14
15
 return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
16
 }
17
18
 Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information inter-
19
 face is finalized.
20
21
 int MPI_Finalize(void)
22
 {
23
 int err;
^{24}
 err=PMPI_T_pvar_handle_free(session, &handle);
25
 err=PMPI_T_pvar_session_free(&session);
26
 err=PMPI_T_finalize();
27
 return PMPI_Finalize();
28
 }
29
30
 14.3.8
 Variable Categorization
^{31}
 MPI implementations can optionally group performance and control variables into categories
32
 to express logical relationships between various variables. For example, an MPI implemen-
33
 tation could group all control and performance variables that refer to message transfers in
34
 the MPI implementation and thereby distinguish them from variables that refer to local
35
 resources such as memory allocations or other interactions with the operating system.
36
 Categories can also contain other categories to form a hierarchical grouping. Categories
37
 can never include themselves, either directly or transitively within other included categories.
38
 Expanding on the example above, this allows MPI to refine the grouping of variables referring
39
 to message transfers into variables to control and to monitor message queues, message
40
 matching activities and communication protocols. Each of these groups of variables would
41
 be represented by a separate category and these categories would then be listed in a single
42
 category representing variables for message transfers.
43
 The category information may be queried in a fashion similar to the mechanism for
44
 querying variable information. The MPI implementation exports a set of N categories via
45
 the MPI tool information interface. If N = 0, then the MPI implementation does not export
46
 any categories, otherwise the provided categories are indexed from 0 to N-1. This index
47
48
```

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.

#### Category Query Functions

The following function can be used to query the number of categories, N.

#### MPI_T_CATEGORY_GET_NUM(num_cat)

OUT num_cat current number of categories (integer)

#### int MPI_T_category_get_num(int *num_cat)

Individual category information can then be queried by calling the following function:

# MPI_T_CATEGORY_GET_INFO(cat_index, name, name_len, desc, desc_len, num_cvars, num_pvars, num_categories)

		,	25
IN	cat_index	index of the category to be queried (integer)	25 26
OUT	name	buffer to return the string containing the name of the category (string)	27 28
INOUT	name_len	length of the string and/or buffer for name (integer)	29
OUT	desc	buffer to return the string containing the description of the category (string)	30 31 32
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)	33
OUT	num_cvars	number of control variables in the category (integer)	34
OUT	num_pvars	number of performance variables in the category (in-teger)	35 36 37
OUT	num_categories	number of categories contained in the category (integer)	38 39

#### 

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.

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

1 If any OUT parameter to MPI_T_CATEGORY_GET_INFO is a NULL pointer, the im- $\mathbf{2}$ plementation will ignore the parameter and not return a value for the parameter. 3 The arguments desc and desc_len are used to return the description of the category as 4 described in Section 14.3.3. 5Returning a description is optional. If an MPI implementation decides not to return a 6 description, the first character for desc must be set to the null character and desc_len must  $\overline{7}$ be set to one at the return of this call. 8 The function returns the number of control variables, performance variables and other 9 categories contained in the queried category in the arguments num_cvars, num_pvars, and 10 num_categories, respectively. 11 If the name of a category is equivalent across connected MPI processes, then the re-12turned description must be equivalent. 13 14MPI_T_CATEGORY_GET_INDEX(name, cat_index) 1516IN the name of the category (string) name 17OUT cat_index the index of the category (integer) 18 19int MPI_T_category_get_index(const char *name, int *cat_index) 2021MPI_T_CATEGORY_GET_INDEX is a function for retrieving the index of a category 22given a known category name. The **name** parameter is provided by the caller, and **cat_index** 23is returned by the MPI implementation. The name parameter is a string terminated with a  24 null character. 25This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME 26 if name does not match the name of any category provided by the implementation at the 27time of the call. 2829 Rationale. This routine is provided to enable fast retrieval of a category index 30 by a tool, assuming it knows the name of the category for which it is looking. The  31 number of categories exposed by the implementation can change over time, so it is not 32 possible for the tool to simply iterate over the list of categories once at initialization. 33 Although using MPI implementation specific category names is not portable across 34 MPI implementations, tool developers may choose to take this route for lower overhead 35 at runtime because the tool will not have to iterate over the entire set of categories 36 to find a specific one. (End of rationale.) 37 38 39 40 41 4243 44 4546 4748

#### Category Member Query Functions

			2
			3 4
		ARS(cat_index, len, indices)	5
IN	cat_index	index of the category to be queried, in the range $[0, N-1]$ (integer)	6 7
IN	len	the length of the indices array (integer)	8
OUT	indices	an integer array of size len, indicating control variable	9
		indices (array of integers)	10 11
int MPI	_T_category_get_cv	vars(int cat_index, int len, int indices[])	12 13
		<b>CVARS</b> can be used to query which control variables are gory. A category contains zero or more control variables.	14 15 16
		ADC(act index law indicac)	17
		ARS(cat_index, len, indices)	18
IN	cat_index	index of the category to be queried, in the range $[0, N-1]$ (integer)	19 20
IN	len	the length of the indices array (integer)	21 22
OUT	indices	an integer array of size len, indicating performance variable indices (array of integers)	22 23 24 25
int MPI	_T_category_get_pv	vars(int cat_index, int len, int indices[])	25 26
	ained in a particular	<b>_PVARS</b> can be used to query which performance variables category. A category contains zero or more performance	27 28 29 30 31
MPI_T_C	CATEGORY_GET_CA	TEGORIES(cat_index, len, indices)	32
IN	cat_index	index of the category to be queried, in the range $[0, N-$	33
		1] (integer)	34
IN	len	the length of the indices array (integer)	35 36
OUT	indices	an integer array of size len, indicating category indices	37
001	malees	(array of integers)	38
			39
int MPI	_T_category_get_ca	tegories(int cat_index, int len, int indices[])	40 41
MPI	T CATEGORY GET	<b>CATEGORIES</b> can be used to query which other categories	41 42
		category. A category contains zero or more other categories.	43
	-	I implementations can grow the number of categories as well	44
		other categories within a category. In order to allow users	45
of the M	PI tool information i	interface to check quickly whether new categories have been	46

Unofficial Draft for Comment Only

added or new variables or categories have been added to a category, MPI maintains a

1 2 3		estamp. This timestamp is more the following function:	notonically increasing during the execution and is
4 5	MPI_T_CAT	<pre>FEGORY_CHANGED(stamp)</pre>	
6 7 8	OUT	stamp	a virtual time stamp to indicate the last change to the categories (integer)
9 10	int MPI_T_	_category_changed(int *st	amp)
11 12 13	the category	y information has not changed	e return the same timestamp, it is guaranteed that between the two calls. If the timestamp retrieved categories have been added or expanded.
14 15 16 17	for ch	-	value is purely virtual and only intended to check tion. It should not be used for any other purpose.
<ol> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> <li>27</li> <li>28</li> <li>29</li> </ol>	MPI_T_CAT as input to MPI_T_CAT The us MPI_T_CAT MPI_T_CAT up to len et function ret	FEGORY_GET_PVARS and M MPI_T_CVAR_GET_INFO, I FEGORY_GET_INFO, respecti er is responsible for allocatin FEGORY_GET_CVARS, MPI_ FEGORY_GET_CATEGORIES. lements into the array. If the curns an arbitrary subset of a	s by MPI_T_CATEGORY_GET_CVARS, PI_T_CATEGORY_GET_CATEGORIES can be used MPI_T_PVAR_GET_INFO and vely. g the arrays passed into the functions T_CATEGORY_GET_PVARS and Starting from array index 0, each function writes e category contains more than len elements, the size len. Otherwise, the entire set of elements is e array, and any remaining array entries are not
30	14.3.9 Re	turn Codes for the MPI Tool	Information Interface
<ol> <li>31</li> <li>32</li> <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> </ol>	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 and follow the same rules and restrictions. In particular, they must satisfy:		
43 44		$0 = MPI_SUCCESS < MPI_T$	$-$ ERR_XXX $\leq$ MPI_ERR_LASTCODE.
45 46 47 48	cause	applications cannot portably	n interface functions must return error classes, be- call MPI_ERROR_CLASS before MPI initialization an error class. ( <i>End of rationale.</i> )

#### 14.3.10 Profiling Interface

All requirements for the profiling interfaces, as described in Section 14.2, also apply to the MPI tool information interface. All rules, guidelines, and recommendations from Section 14.2 apply equally to calls defined as part of the MPI tool information interface.

Return Code	Description
Return Codes for All Functions in t	he MPI Tool Information Interface
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid use of the interface or bad parameter
	values(s)
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
MPI_T_ERR_INVALID_ITEM	The item index queried is out of range
	(for MPI_T_ENUM_GET_ITEM only);
	This error code is deprecated in MPI-3.2 and
	superseded by MPI_T_ERR_INVALID_INDEX
Roturn Codes for Variable and Cata	gory Query Functions: MPI_T_*_GET_*
	The variable or category index is invalid
MPI_T_ERR_INVALID_INDEX	0.0
MPI_T_ERR_INVALID_NAME	The variable or category name is invalid
Return Codes for Handle Functions:	
MPI_T_ERR_INVALID_INDEX	The variable index is invalid
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_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 atomicall
	(for MPI_T_PVAR_READRESET)
Return Codes for Category Function	
MPI_T_ERR_INVALID_INDEX	The category index is invalid

## Chapter 15

## **Deprecated Interfaces**

#### Deprecated since MPI-2.0 15.1

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 18.2.7. The language bindings are modified.

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for keyval	23
IIN		Copy canback function for Reyval	24
IN	delete_fn	Delete callback function for keyval	25
OUT	keyval	key value for future access (integer)	26
IN	extra_state	Extra state for callback functions	27
			28
int MDT K	eyval_create(MPI_Copy_fun	ction *conv fn	29
IIIC MFI_K	• • • •		30
		<pre>*delete_fn, int *keyval,</pre>	31
	void* extra_state)		32
For this rou	utine, an interface within the	mpi_f08 module was never defined.	33
			34
	•	FN, KEYVAL, EXTRA_STATE, IERROR)	35
	NAL COPY_FN, DELETE_FN		36
INTEG	ER KEYVAL, EXTRA_STATE, I	ERRUR	37
The co	py_fn function is invoked wh	nen a communicator is duplicated by	38
MPI_COMM_DUP. copy_fn should be of type MPI_Copy_function, which is defined as follows:			39
-	- 13-		40
			41
typedef i	nt MPI_Copy_function(MPI_	Comm oldcomm, int keyval,	42

typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag) A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi_f08 module was never defined.

```
626
 CHAPTER 15. DEPRECATED INTERFACES
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	_val)	1
INOUT	comm	communicator to which attribute will be attached (handle)	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, ir	t keyval, void* attribute_val)	9 10
For this ro	utine, an interface within the	mpi_f08 module was never defined.	11
	PUT(COMM, KEYVAL, ATTRIBU ER COMM, KEYVAL, ATTRIBUT	-	12 13 14
MPI-2.0. 1	The language independent defi	and is superseded by MPI_COMM_GET_ATTR in nition of the deprecated function is the same as of name. The language bindings are modified.	15 16 17 18
MPI_ATTR	R_GET(comm, keyval, attribute	_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	25 26 27
int MPI_A	.ttr_get(MPI_Comm comm, ir	nt keyval, void *attribute_val, int *flag)	28
For this ro	utine, an interface within the	mpi_f08 module was never defined.	29 30
INTEG	GET(COMM, KEYVAL, ATTRIBU ER COMM, KEYVAL, ATTRIBUT CAL FLAG		31 32 33
in MPI-2.0 $$	. The language independent d	and is superseded by MPI_COMM_DELETE_ATTR efinition of the deprecated function is the same as on name. The language bindings are modified.	34 35 36 37 38
MPI_ATTF	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)	42
			$43 \\ 44$
int MPI_A	ttr_delete(MPI_Comm comm,	int keyval)	44 45
For this ro	utine, an interface within the	mpi_f08 module was never defined.	46
MPI_ATTR_	DELETE(COMM, KEYVAL, IERF	COR)	47 48

### Unofficial Draft for Comment Only

1 INTEGER COMM, KEYVAL, IERROR  $\mathbf{2}$ 3 4 15.2 Deprecated since MPI-2.2 5The entire set of C++ language bindings have been removed. See Chapter 16, Removed 6 Interfaces for more information.  $\overline{7}$ The following function typedefs have been deprecated and are superseded by new 8 names. Other than the typedef names, the function signatures are exactly the same; the 9 names were updated to match conventions of other function typedef names. 1011Deprecated Name New Name 12MPI_Comm_errhandler_fn MPI_Comm_errhandler_function 13 MPI_File_errhandler_fn MPI_File_errhandler_function 14MPI_Win_errhandler_fn MPI_Win_errhandler_function 15161715.3 Deprecated since MPI-3.2 18 Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed 19in a future version of the MPI specification. 2021The following return class has been deprecated and is superseded by a new name. 22Deprecated Name Replacement Name 23MPI_T_ERR_INVALID_ITEM MPI_T_ERR_INVALID_INDEX  24 2526272829 30  31 32 33 3435 36 37 38 39 40414243 444546 4748

### 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_ADDRESSMPI_GET_ADDRESSMPI_ERRHANDLER_CREATEMPI_COMM_CREATE_ERRHANDLERMPI_ERRHANDLER_GETMPI_COMM_GET_ERRHANDLERMPI_ERRHANDLER_SETMPI_COMM_SET_ERRHANDLERMPI_TYPE_EXTENTMPI_TYPE_GET_EXTENTMPI_TYPE_HINDEXEDMPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI_TYPE_CREATE_HVECTORMPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCTMPI_TYPE_UBMPI_TYPE_GET_EXTENT	Removed	MPI-2 Replacement
MPI_ERRHANDLER_GETMPI_COMM_GET_ERRHANDLERMPI_ERRHANDLER_SETMPI_COMM_SET_ERRHANDLERMPI_TYPE_EXTENTMPI_TYPE_GET_EXTENTMPI_TYPE_HINDEXEDMPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI_TYPE_CREATE_HVECTORMPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCT	MPI_ADDRESS	MPI_GET_ADDRESS
MPI_ERRHANDLER_SETMPI_COMM_SET_ERRHANDLERMPI_TYPE_EXTENTMPI_TYPE_GET_EXTENTMPI_TYPE_HINDEXEDMPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI_TYPE_CREATE_HVECTORMPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCT	MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_TYPE_EXTENTMPI_TYPE_GET_EXTENTMPI_TYPE_HINDEXEDMPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI_TYPE_CREATE_HVECTORMPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCT	MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_TYPE_HINDEXEDMPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI_TYPE_CREATE_HVECTORMPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCT	MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_TYPE_HVECTORMPI_TYPE_CREATE_HVECTORMPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCT	MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_LBMPI_TYPE_GET_EXTENTMPI_TYPE_STRUCTMPI_TYPE_CREATE_STRUCT	MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_STRUCT MPI_TYPE_CREATE_STRUCT	MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
	MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB 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 shows the removed MPI-1 datatypes and their replacements.

#### 16.1.4 Removed MPI-1 Constants

Table 16.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

 $44 \\ 45$ 

	630 CHAPTER 16. REMOVED INTERFACES
1	Removed MPI-2 Replacement
2	MPI_LB MPI_TYPE_CREATE_RESIZED
3	MPI_UB MPI_TYPE_CREATE_RESIZED
4	
5	
6	Table 16.2: Removed MPI-1 datatypes and their replacements
7	Removed MPI-1 Constants
8	C type: const int (or unnamed enum)
9 10	Fortran type: INTEGER
11	MPI_COMBINER_HINDEXED_INTEGER
12	MPI_COMBINER_HVECTOR_INTEGER
13	MPI_COMBINER_STRUCT_INTEGER
14	
15	Table 16.3: Removed MPI-1 constants
16	
17 18	16.1.5 Removed MPI-1 Callback Prototypes
19	Table 16.4 shows the removed MPI-1 callback prototypes and their MPI-2 replacements.
20	
21	Removed MPI-2 Replacement
22	MPI_Handler_function MPI_Comm_errhandler_function
23	
24	Table 16.4: Removed MPI-1 callback prototypes and their replacements
25 26	
20 27	
28	16.2 C++ Bindings
29	The C++ bindings were deprecated as of MPI-2.2. The C++ bindings are removed in
30	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by
31	an implementation as described in the MPI-2.2 standard.
32	•
33 34	
35	
36	
37	
38	
39	
40	
41	
42 43	
44	
45	
46	
47	
48	

Chapter 17			
Backward Incompatibilities			
17.1 Backward Incompatible since MPI-3.2 The default communicator where errors are raised when not involving a communicator, window, or file was changed from MPI_COMM_WORLD to MPI_COMM_SELF.			

### Unofficial Draft for Comment Only

### Chapter 18

# Language Bindings

#### 18.1 Fortran Support

#### 18.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [40] + TS 29113 [41].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 18.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi_f08: This method is described in Section 18.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 18.1.3 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 18.1.4. 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.

Compliant MPI-3 implementations providing a Fortran interface must provide one or both of the following:

- The USE mpi_f08 Fortran support method.
- The USE mpi and INCLUDE 'mpif.h' Fortran support methods.

Section 18.1.6 describes restrictions if the compiler does not support all the needed features. Application subroutines and functions may use either one of the modules or the mpif.h include file. An implementation may require the use of one of the modules to prevent type mismatch errors.

Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h enforces type checking on a particular system. Using a module provides several potential advantages over using an include file; the mpi_f08 module offers the most robust and complete Fortran support. (*End of advice to users.*)

In a single application, it must be possible to link together routines which USE mpi_f08, USE mpi, and INCLUDE 'mpif.h'.

The LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED is set to

19.TRUE. if all buffer choice arguments are defined in explicit interfaces with assumed-type 20and assumed-rank [41]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22 added to the choice buffer arguments of all nonblocking interfaces and the underlying 23Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of  24 TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 25method, but not in the C header file. The values may be different for each Fortran support 26method. All other constants and the integer values of handles must be the same for each 27Fortran support method. 28

Section 18.1.2 through 18.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 18.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for  31 the profiling interface. Section 18.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 18.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 18.1.8 and Section 18.1.9 de-34scribe additional functionality that is part of the Fortran support. MPI_F_SYNC_REG is 35 needed for one of the methods to prevent register optimization problems. A set of func-36 tions provides additional support for Fortran intrinsic numeric types, including parameter-37 ized types: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, 38 MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. In the context 39 of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 40 parameters. Sections 18.1.10 through 18.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 18.1.20 compares the Fortran problems 42with those in C.

43 44

45

#### 18.1.2 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 18.1.6 describes restrictions if the compiler
 does not support all the needed features. Within all MPI function specifications, the first

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4 5

6 7

8

9

10

11

12

13

14

15 16

17

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 18.1.3.

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 the derived type MPI_Status, and define all MPI handles with uniquely named handle types (instead of INTEGER handles, as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators .EQ. and .NE. to allow the comparison of these MPI handles with .EQ., .NE., == and /=.
- Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations, and set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113). See Section 18.1.6 for older compiler versions.
- Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., TYPE(*), DIMENSION(..) in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

*Rationale.* In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TS 29113 feature is not needed for the support of non-contiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 18.1.6 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

 $\frac{4}{5}$ 

 31 

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) as input, an INTENT is not specified. (End of rationale.)

Advice to users. If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [40], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of INTENT(OUT) the copy-in may be suppressed by the optimization and the routine 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 18.1.3. (End of advice to users.)

• Declare all ierror output arguments as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN).

*Rationale.* For user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the mpi_f08 module are designed based on the Fortran 2008 standard [40] together with the Technical Specification "TS 29113 Further Interoperability with C" [41] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [41], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the mpi_f08 module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., REAL, CHARACTER*5, CHARACTER*(*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi_f08 module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

#### 18.1.3 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. If an implementation is paired with a compiler that either does not support TYPE(*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 18.1.6 for more details.
- 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.

*Rationale.* They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (*End of rationale.*)

- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi_f08 module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces and the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to .FALSE..

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

For an MPI implementation that fully supports nonblocking calls Advice to users. with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copy-in/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 18.1.12 for more details. (End of advice to users.)

- A high quality MPI implementation may enhance the interface by using TYPE(*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !\$PRAGMA IGNORE_TKR or a set of overloaded functions as described by M. Hennecke in [28], if the compiler supports this TS 29113 language feature. See Section 18.1.6 for further details.
  - Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(*), DIMENSION(..), otherwise set it to .FALSE.. When MPI_SUBARRAYS_SUPPORTED is defined as
    - .TRUE., non-contiguous sub-arrays can be used as buffers in nonblocking routines.
  - Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in non-blocking calls may be disallowed. See Section 18.1.6 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.

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
 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

#### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

was changed in several places in MPI-2. For instance, MPI_IN_PLACE changes the intent of an OUT argument to be INOUT. (End of rationale.)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include mpif.h may not expect that 11 INTENT (OUT) is used. In particular, output array arguments are expected to keep their 12content as long as the MPI routine does not modify them. To keep this behavior, it is 13 recommended that implementations not use INTENT(OUT) in the mpi module and the 14mpif.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 18.1.4

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:

- Define all named MPI constants. • Declare MPI functions that return a value. • Define all handles as INTEGER. • Be valid and equivalent for both fixed and free source form. For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted). • Set the LOGICAL compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the mpi module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to .FALSE.. Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is strongly encouraged for the following reasons: • Most mpif.h implementations do not include compile-time argument checking. • Therefore, many bugs in MPI applications remain undetected at compile-time, such as:
  - Missing ierror as last argument in most Fortran bindings.

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

1  $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

15

16 17 18

19

20

21

22

23

24

2526

2728

29

30 31

32 33

34

35

36

37

38

39 40

41

4243

44

45

46 47

640 CHAPTER 18. LANGUAGE BINDINGS
<ul> <li>Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.</li> <li>Incorrect argument positions; e.g., interchanging the count and</li> </ul>
datatype arguments.
- Passing incorrect MPI handles; e.g., passing a datatype instead of a commu-
nicator.
• The migration from mpif.h to the mpi module should be relatively straightforward (i.e., substituting include 'mpif.h' after an implicit statement by use mpi before that implicit statement) as long as the application syntax is correct.
• Migrating portable and correctly written applications to the <b>mpi</b> module is not expected to be difficult. No compile or runtime problems should occur because an <b>mpif.h</b> include file was always allowed to provide explicit Fortran interfaces.
(End of advice to users.)
<i>Rationale.</i> With MPI-3.0, the mpif.h include file was not deprecated in order to retain strong backward compatibility. Internally, mpif.h and the mpi module may be implemented so that essentially the same library implementation of the MPI routines can be used. ( <i>End of rationale.</i> )
18.1.5 Interface Specifications, Procedure Names, and the Profiling Interface
The Fortran interface specification 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 Fortran standard allows a given Fortran interface to be implemented with several methods, e.g., within or outside of a module, with or without BIND(C), or the buffers with or without TS 29113. Such implementation decisions imply different binary interfaces and different specific procedure names. The requirements for several implementation schemes together with the rules for the specific procedure names and its implications for the profiling interface are specified within this section, but not the implementation details.
<i>Rationale.</i> This section was introduced in MPI-3.0 on Sep. 21, 2012. The major goals for implementing the three Fortran support methods have been:
• Portable implementation of the wrappers from the MPI Fortran interfaces to the MPI routines in C.
• Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE
• The Fortran PMPI interface need not be backward compatible, but a method must be included that a tools layer can use to examine the MPI library about the specific procedure names and interfaces used.
• No performance drawbacks.
• Consistency between all three Fortran support methods.
• Consistent with Fortran $2008 + TS 29113$ .

No.	Specific pro- cedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.4, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like !\$PRAGMA IGNORE_TKR, which provides a call-by-reference argument without type, kind, and dimension checking.
1B	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.4, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION().
2A	MPI_ISEND	Fortran interface and arguments, as in Annex A.5, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like !\$PRAGMA IGNORE_TKR, which provides a call-by-reference argument without type, kind, and dimension checking.
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.5, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION().

Table 18.1: Specific Fortran procedure names and related calling conventions. MPI_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The LOGICAL arguments are not interoperable with C, mainly because the internal representations for .FALSE. and .TRUE. are compiler dependent. The provided interface was mainly based on BIND(C) interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the BIND(C) had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI_COMM_DUP_FN. Non-BIND(C) procedures are also not interoperable with C, and therefore the BIND(C) had to be removed from all routines with PROCEDURE arguments, e.g., from MPI_OP_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (End of rationale.)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 18.1. Case is not significant in the names.

Note that for the deprecated routines in Section 15.1, which are reported only in Annex A.5, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi_f08 module.

To set MPI_SUBARRAYS_SUPPORTED to .TRUE. within a Fortran support method, it is required that all non-blocking and split-collective routines with buffer arguments are

#### Unofficial Draft for Comment Only

implemented according to 1B and 2B, i.e., with MPI_Xxxx_f08ts in the mpi_f08 module,
 and with MPI_XXXX_FTS in the mpi module and the mpif.h include file.

The mpi and mpi_f08 modules and the mpif.h include file will each correspond to exactly one implementation scheme from Table 18.1. However, the MPI library may contain multiple implementation schemes from Table 18.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

10 Rationale. After a compiler provides the facilities from TS 29113, i.e., TYPE(*), 11 DIMENSION(...), it is possible to change the bindings within a Fortran support method 12to support subarrays without recompiling the complete application provided that the 13 previous interfaces with their specific procedure names are still included in the li-14brary. Of course, only recompiled routines can benefit from the added facilities. 15There is no binary compatibility conflict because each interface uses its own spe-16cific procedure names and all interfaces use the same constants (except the value of 17 MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type 18 definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 19 operations can be protected through the ASYNCHRONOUS attribute, and the proce-20dure declarations in the mpi_f08 and mpi module and the mpif.h include file declare 21choice buffers with the ASYNCHRONOUS attribute, then the value of 22

- MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to .TRUE. in the module definition and include file. (*End of rationale.*)
  - Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is and what is not supported. (*End of advice to users.*)

28 29

25

26

27

Within the mpi_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI_lsend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

³⁵ A user-written or middleware profiling routine should provide the same specific Fortran ³⁶ procedure names and calling conventions, and therefore can interpose itself as the MPI ³⁷ 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, choice buffer arguments, or that are attribute caching routines (

MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should
 invoke the corresponding PMPI routine using the same Fortran support method as used in
 the calling application program, because the C, mpi_f08 and mpi callback prototypes are
 different or the meaning of the choice buffer or attribute_val arguments are different.

- ⁴⁵ Advice to users. Although for each support method and MPI routine (e.g.,
- ⁴⁶ MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept ⁴⁷ the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), ⁴⁸ each profiling routine itself uses only one support method (e.g., mpi_f08) and calls

#### Unofficial Draft for Comment Only

7

8

the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI_lsend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

```
SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

INTERFACE PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)

with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the mpi_f08 module.

For example, the interface specifications together with the specific procedure names can be implemented with

#### Unofficial Draft for Comment Only

 31 

```
1
 MODULE mpi_f08
2
 TYPE, BIND(C) :: MPI_Comm
3
 INTEGER :: MPI_VAL
 END TYPE MPI_Comm
4
 . . .
5
 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
6
 SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
7
 IMPORT :: MPI_Comm
8
 TYPE(MPI_Comm),
 INTENT(IN) :: comm
9
 INTENT(OUT) :: rank
 INTEGER,
 INTEGER, OPTIONAL,
 INTENT(OUT) :: ierror
10
 END SUBROUTINE
11
 END INTERFACE
12
 END MODULE mpi_f08
13
14
 MODULE mpi
15
 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
16
 SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
17
 INTEGER, INTENT(IN) :: comm
 ! The INTENT may be added although
 INTEGER, INTENT(OUT) :: rank
 ! it is not defined in the
18
 INTEGER, INTENT(OUT) :: ierror ! official routine definition.
19
 END SUBROUTINE
20
 END INTERFACE
21
 END MODULE mpi
22
23
 And if interfaces are provided in mpif.h, they might look like this (outside of any
24
 module and in fixed source format):
25
 123456789012
26
 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
27
 SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
28
 INTEGER, INTENT(IN) :: comm ! The argument names may be
29
 INTEGER, INTENT(OUT) :: rank
 ! shortened so that the
30
 INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
31
 END SUBROUTINE
 ! maximum of 72 characters.
32
 END INTERFACE
33
34
 (End of advice to implementors.)
35
 The following is an example of how a user-written or middleware
 Advice to users.
36
 profiling routine can be implemented:
37
38
 SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
39
 USE :: mpi_f08, my_noname => MPI_Isend_f08ts
40
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
41
 INTEGER,
 INTENT(IN)
 :: count, dest, tag
42
 TYPE(MPI_Datatype), INTENT(IN)
 :: datatype
 TYPE(MPI_Comm),
 INTENT(IN)
 :: comm
43
 TYPE(MPI_Request), INTENT(OUT)
 :: request
44
 INTEGER, OPTIONAL,
 INTENT(OUT)
 :: ierror
45
 ! ... some code for the begin of profiling
46
 call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
47
 ! ... some code for the end of profiling
48
 END SUBROUTINE MPI_Isend_f08ts
```

Unofficial Draft for Comment Only

Note that this routine is used to intercept the existing specific procedure name MPI_lsend_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI_lsend. The USE of the mpi_f08 module is needed for definitions of handle types and the interface for PMPI_lsend. However, this module also contains an interface definition for the specific procedure name MPI_lsend_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the USE here specifically excludes the interface from the module by renaming the unused routine name in the mpi_f08 module into "my_noname" in the scope of this routine. (*End of advice to users.*)

The PMPI interface allows intercepting MPI routines. For exam-Advice to users. 11 ple, an additional MPI_ISEND profiling wrapper can be provided that is called by the 12application and internally calls PMPI_ISEND. There are two typical use cases: a pro-13 filing layer that is developed independently from the application and the MPI library. 14and profiling routines that are part of the application and have access to the appli-15cation data. With MPI-3.0, new Fortran interfaces and implementation schemes were 16 introduced that have several implications on how Fortran MPI routines are internally 17 implemented and optimized. For profiling layers, these schemes imply that several in-18 ternal interfaces with different specific procedure names may need to be intercepted, 19 as shown in the example code above. Therefore, for wrapper routines that are part 20of a Fortran application, it may be more convenient to make the name shift within 21the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a 22call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to 23call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface.  24 (End of advice to users.) 25

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 18.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

#### 18.1.6 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 may be up to 30 characters (31 with the profiling interface).
  - 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 by 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 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.)
45
46
47
48

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10

26

27

28

29

30 31

32 33

34

35

36 37

38

39 40

41

42

43

1	• For Fortran 90:
2	The major additional features that are needed from Fortran 90 are:
3 4	- The MODULE and INTERFACE concept.
5	- The KIND= and SELECTEDKIND concept.
6	- Fortran derived TYPEs and the SEQUENCE attribute.
7	
8	- The OPTIONAL attribute for dummy arguments.
9 10	<ul> <li>Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.</li> </ul>
11	With these features, $MPI-1.1 - MPI-2.2$ can be implemented without restrictions.
12 13	MPI-3.0 can be implemented with some restrictions. The Fortran support methods
14	are abbreviated with $S1={\tt the\ mpi_f08\ module},S2={\tt the\ mpi\ module},{\tt and}\ S3={\tt the}$
15	mpif.f include file. If not stated otherwise, restrictions exist for each method which
16	prevent implementing the complete semantics of MPI-3.0.
17	- MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-
18	contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,
19	or split-collective I/O.
20	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa-
21 22	tion is possible.
23	- In this preliminary interface of S1, the following changes are necessary:
24	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
25	like !\$PRAGMA IGNORE_TKR.
26	* The ASYNCHRONOUS attribute is omitted.
27 28	* <b>PROCEDURE()</b> callback declarations are substituted by <b>EXTERNAL</b> .
29	- The specific procedure names are specified in Section 18.1.5.
30	- Due to the rules specified in Section 18.1.5, choice buffer declarations should be
31	implemented only with non-standardized extensions like <b>!\$PRAGMA IGNORE_TKR</b>
32	(as long as $F2008+TS$ 29113 is not available).
33	In S2 and S3: Without such extensions, routines with choice buffers should be
34	provided with an implicit interface, instead of overloading with a different MPI
35	function for each possible buffer type (as mentioned in Section $18.1.11$ ). Such
36 37	overloading would also imply restrictions for passing Fortran derived types as
38	choice buffer, see also Section $18.1.15$ .
39	Only in S1: The implicit interfaces for routines with choice buffer arguments
40	imply that the ierror argument cannot be defined as OPTIONAL. For this reason,
41	it is recommended not to provide the mpi_f08 module if such an extension is not
42	available.
43	- The ASYNCHRONOUS attribute can <b>not</b> be used in applications to protect buffers
44	in nonblocking MPI calls $(S1-S3)$ .
45	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE
46	routines is not available.
47 48	

	<ul> <li>In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types related to TYPE(MPI_Status) (see Section 18.2.5) are also not available in the mpi module and mpif.h.</li> </ul>	1 2 3 4 5 6 7 8 9
•	For Fortran 95: The quality of the MPI interface and the restrictions are the same as with Fortran 90.	10 11 12
•	For Fortran 2003: The major features that are needed from Fortran 2003 are:	13 14 15
	<ul> <li>Interoperability with C, i.e.,</li> <li>* BIND(C) derived types.</li> </ul>	16 17 18
	<ul> <li>* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.</li> <li>The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.</li> </ul>	19 20 21
	<ul> <li>The ability to overload the operators .EQ. and .NE. to allow the comparison of derived types (used in MPI-3.0 for MPI handles).</li> </ul>	22 23 24
	<ul> <li>The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.</li> </ul>	24 25 26 27
	With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:	28 29 30
	<ul> <li>The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a void * argument.</li> </ul>	31 32 33 34
	MPI-3.0 can be implemented with the following restrictions:	35 36
	- MPI_SUBARRAYS_SUPPORTED equals .FALSE	37 38
	<ul> <li>For S1, only a preliminary implementation is possible. The following changes are necessary:</li> </ul>	39 40
	* TYPE(*), DIMENSION() is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.	41 42
	- The specific procedure names are specified in Section 18.1.5.	43
	<ul> <li>With S1, the ASYNCHRONOUS is required as specified in the second Fortran inter- faces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.</li> </ul>	44 45 46

1 2 3	<ul> <li>The ASYNCHRONOUS Fortran attribute can be used in applications to try to 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</li> </ul>
4	asynchronous Fortran I/O and $MPI$ communication.
5 6	<ul> <li>The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can</li> </ul>
7	be used only for Fortran types that are C compatible.
8 9	<ul> <li>The same restriction as for Fortran 90 applies if non-standardized extensions like</li></ul>
10 11	For Fortran $2008 + TS 29113$ and later and
12	For Fortran $2003 + TS 29113$ and fater and For Fortran $2003 + TS 29113$ :
13	The major feature that are needed from TS 29113 are:
14	·
15	- TYPE(*), DIMENSION() is available.
16 17	<ul> <li>The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com- munication.</li> </ul>
18	- The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not
19	restricted to Fortran types for which a corresponding type in C exists.
20	Using these features, MPI-3.0 can be implemented without any restrictions.
21	
22	- With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE. The ASYNCHRONOUS at-
23	tribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR)
24	binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
25 26	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used for any Fortran type.
27	<ul> <li>With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation</li> </ul>
28	dependent. A high quality implementation will also provide
29	MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the
30	ASYNCHRONOUS attribute in the same way as in S1.
31	- If non-standardized extensions like <b>!</b> \$PRAGMA IGNORE_TKR are not available then
32	S2 must be implemented with TYPE(*), DIMENSION().
33	-
34	Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice
35	argument may be implemented with an explicit interface using compiler directives, for example:
36	ior example.
37	INTERFACE
38	SUBROUTINE MPI(buf,)
39	!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf
40 41	!\$PRAGMA IGNORE_TKR buf
42	!DIR\$ IGNORE_TKR buf
43	!IBM* IGNORE_TKR buf
44	REAL, DIMENSION(*) :: buf
45	! declarations of the other arguments
46	END SUBROUTINE
47	END INTERFACE
48	(End of advice to implementors.)

#### 18.1.7 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 18.1.11 through 18.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi_f08 and mpi modules, and mpif.h) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [41] are available. This is required only for mpi_f08. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the mpi_f08 module with MPI_SUBARRAYS_SUPPORTED set to .FALSE..
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 18.1.12 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 protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==.FALSE.. Observation of these rules by the MPI application developer is especially recomended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows:  $\overline{7}$ 

 24 

1 2 3 4	• 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 672 and Section 18.1.8, and DD on page 673) solve the problems described in Section 18.1.17.
5 6 7 8 9	• The problems with temporary data movement (described in detail in Section 18.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
10 11 12 13 14	• Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 18.1.19) are resolved <b>without</b> any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.
15 16	All of these rules are valid for the mpi_f08 and mpi modules and independently of whether mpif.h uses explicit interfaces.
17 18 19 20 21	Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [41], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (End of advice to implementors.)
22 23	18.1.8 Additional Support for Fortran Register-Memory-Synchronization
24 25 26 27	As described in Section 18.1.17, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.
28 29	MPI_F_SYNC_REG(buf)
30 31	INOUT buf initial address of buffer (choice)
32 33 34	<pre>MPI_F_sync_reg(buf)     TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf</pre>
35 36	<pre>MPI_F_SYNC_REG(buf)</pre>
37 38 39 40 41	This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.
42 43 44 45	<i>Rationale.</i> 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. ( <i>End of rationale.</i> )
46 47 48	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 muttine can be remained as part of an antimization). However, it is

#### Unofficial Draft for Comment Only

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 be called if MPI_ASYNC_PROTECTS_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (*End of advice to users*.)

# 18.1.9 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.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER) with an optional integer KIND parameter that selects from among one or more variants. The specific meaning of different KIND values themselves are implementation dependent and not specified by the language. Fortran provides the KIND selection functions selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two declarations are equivalent:

double precision x
real(KIND(0.0d0)) x

MPI provides two orthogonal methods for handling communication buffers of numeric intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method (see "Support for size-specific MPI Datatypes" on page 655) gives the user complete control over communication by exposing machine representations.

 $46 \\ 47$ 

48

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16 17

18 19

20

21

22

23

 24 

25

26

27

28

29

30

 31 

32

33

34

35 36

37

38

39

40

41

42

43

44

 $\frac{1}{2}$ 

652

## Parameterized Datatypes with Specified Precision and Exponent Range

 $_{\scriptscriptstyle 3}$   $\,$   $\,$  MPI provides named data types corresponding to standard Fortran 77 numeric types:

⁴ MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and

⁵ MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides rep-⁶ resentation conversion in heterogeneous environments. The mechanism described in this ⁷ section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables 8 are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND 9 parameter, where  $\mathbf{p}$  is decimal digits of precision and  $\mathbf{r}$  is an exponent range. Implicitly 10 MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 11 defined for each value of (p, r) supported by the compiler, including pairs for which one 12value is unspecified. Attempting to access an element of the array with an index (p, r) not 13 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 14 datatypes. For integers, there is a similar implicit array related to selected_int_kind and 15 indexed by the requested number of digits r. Note that the predefined datatypes contained 16 in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but 17a new set. 18

- Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (End of advice to implementors.)
- Advice to users. selected_real_kind() maps a large number of (p,r) pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same (p,r) value (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (End of advice to users.)
- 31 32

19

20

21

22 23

24

25

26

27

28

29

30

```
33
34
```

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

35	IN	р	precision, in decimal digits (integer)
36	IN	r	decimal exponent range (integer)
37			decimar exponent range (meeger)
38	OUT	newtype	the requested $MPI$ data type (handle)
39			
40 41	int MPI_T	<pre>'ype_create_f90_real(int p</pre>	o, int r, MPI_Datatype *newtype)
41	NDT T		
42	MP1_Type_	_create_f90_real(p, r, new	vtype, lerror)
43	INTEG	ER, INTENT(IN) :: p, r	
44	TYPE(	[MPI_Datatype), INTENT(OU)	[) :: newtype
45	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror
46	ΜΡΤ ΤΥΡΕ	CREATE_F90_REAL(P, R, NEW	TYPE, TERROR)
47		ER P, R, NEWTYPE, IERROR	,,,
48	111100		

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 655.

It is erroneous to supply values for p and r not supported by the compiler.

MPI_TYI	PE_CREATE_F90_0	COMPLEX(p, r, newtype)	11 12
IN	р	precision, in decimal digits (integer)	13
IN	r	decimal exponent range (integer)	14
	-		15
OUT	newtype	the requested MPI datatype (handle)	16
	_		17
int MPL	_Type_create_f90	<pre>)_complex(int p, int r, MPI_Datatype *newtype)</pre>	18
MPI_Type	e_create_f90_com	plex(p, r, newtype, ierror)	19 20
	EGER, INTENT(IN)	•	20
TYPI	E(MPI_Datatype),	INTENT(OUT) :: newtype	21
INTE	EGER, OPTIONAL,	INTENT(OUT) :: ierror	23
MPT TYPE	E CREATE F90 COM	IPLEX(P, R, NEWTYPE, IERROR)	24
	EGER P, R, NEWTY		25
			26
		a predefined MPI datatype that matches a	27
		elected_real_kind(p, r). Either p or r may be omitted from	- 28
		nd(p, r) (but not both). Analogously, either p or r may be se	- 29
		ing rules for datatypes created by this function are analogous t	30
	-	types created by MPI_TYPE_CREATE_F90_REAL. Restriction atype with the "external32" data representation are given o	31
page $655$		atype with the externalsz data representation are given o	32
		ly values for $p$ and $r$ not supported by the compiler.	33
10 15	erroneous to supp	iy values for p and r not supported by the complet.	34
			35
MPI_TYI	PE_CREATE_F90_I	INTEGER(r, newtype)	36
IN	r	decimal exponent range, i.e., number of decimal digit	37 ts
		(integer)	30
OUT	nowtypo		39
001	newtype	the requested MPI datatype (handle)	40
·	ш . соо		41 42
int MPI	_lype_create_190	)_integer(int r, MPI_Datatype *newtype)	42
MPI_Type	e_create_f90_int	eger(r, newtype, ierror)	40
INTE	EGER, INTENT(IN)	:: r	45
	• -	INTENT(OUT) :: newtype	46
INTE	EGER, OPTIONAL,	INTENT(OUT) :: ierror	47
MPI TYPI	E CREATE F90 INT	TEGER(R, NEWTYPE, IERROR)	48

Unofficial Draft for Comment Only

 $1 \\ 2$ 

 $\mathbf{5}$ 

1	INTEGER R, NEWTYPE, IERROR
2 3	This function returns a predefined MPI datatype that matches a INTEGER variable of
4	KIND selected_int_kind(r). Matching rules for datatypes created by this function are
5	analogous to the matching rules for data types created by $MPI_TYPE_CREATE_F90_REAL.$
6	Restrictions on using the returned datatype with the "external32" data representation are
7	given on page $655$ .
8	It is erroneous to supply a value for r that is not supported by the compiler.
9	Example:
10	integer longtype, quadtype
11	integer, parameter :: long = selected_int_kind(15)
12	<pre>integer(long) ii(10)</pre>
13	<pre>real(selected_real_kind(30)) x(10)</pre>
14 15	call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
16	call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
17	•••
18	call MPI_SEND(ii, 10, longtype,)
19	call MPI_SEND(x, 10, quadtype,)
20	
21	Advice to users. The datatypes returned by the above functions are predefined
22	datatypes. They cannot be freed; they do not need to be committed; they can be
23 24	used with predefined reduction operations. There are two situations in which they
25	behave differently syntactically, but not semantically, from the MPI named predefined
26	datatypes.
27	1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to
28	retrieve the values of $p$ and $r$ .
29	2. Because the datatypes are not named, they cannot be used as compile-time
30	initializers or otherwise accessed before a call to one of the
31	MPI_TYPE_CREATE_F90_XXX routines.
32 33	If a variable was declared specifying a non-default KIND value that was not obtained
34	with selected_real_kind() or selected_int_kind(), the only way to obtain a
35	matching MPI datatype is to use the size-based mechanism described in the next
36	section.
37	(End of advice to users.)
38	
39	Advice to implementors. An application may often repeat a call to
40	MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The appli-
41	cation is not allowed to free the returned predefined, unnamed datatype handles. To
42 43	prevent the creation of a potentially huge amount of handles, a high quality MPI imple- mentation should return the same datatype handle for the same (REAL/COMPLEX/
43	INTEGER, p, r) combination. Checking for the combination $(p, r)$ in the preceding call
45	to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated
46	handles should limit the overhead of finding a previously generated datatype with
47	same combination of (XXX,p,r). (End of advice to implementors.)
48	

*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) or user-defined (Section 13.5.3) 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.

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	f		(p	>	33)	or	(r	>	4931)	then	external32 representation
											is undefined
el	lse	if	(p	>	15)	or	(r	>	307)	then	external32_size = 16
el	lse	if	(p	>	6)	or	(r	>	37)	then	external32_size = 8
el	lse										external32_size = 4
For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.											

For MPI_TYPE_CREATE_F90_INTEGER:

if (r > 38) the	n external32 representation is undefined
else if $(r > 18)$ the	n external32_size = 16
else if $(r > 9)$ the	n external32_size = 8
else if $(r > 4)$ the	n external32_size = 4
else if $(r > 2)$ the	n external32_size = 2
else	external32_size = 1

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions, when the "external32" data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

#### Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is  47  a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler,  48 

# Unofficial Draft for Comment Only

 $\overline{7}$ 

 $41 \\ 42$ 

 $45 \\ 46$ 

1 MPI must provide a named size-specific datatype. The name of this datatype is of the form  $\mathbf{2}$ MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, 3 and  $\mathbf{n}$  is the length in bytes of the machine representation. This datatype locally matches 4 all variables of type (typeclass, n) in Fortran. The list of names for such types includes: 5MPI_REAL4 6 MPI_REAL8 7 MPI_REAL16 8 MPI_COMPLEX8 9 MPI_COMPLEX16 10 MPI_COMPLEX32 11 MPI_INTEGER1 12MPI_INTEGER2 13 MPI_INTEGER4 14 MPI_INTEGER8 15MPI_INTEGER16 1617 One datatype is required for each representation supported by the Fortran compiler. 18 19Rationale. Particularly for the longer floating-point types, C and Fortran may use 20different representations. For example, a Fortran compiler may define a 16-byte REAL 21type with 33 decimal digits of precision while a C compiler may define a 16-byte 22 long double type that implements an 80-bit (10 byte) extended precision floating point 23value. Both of these types are 16 bytes long, but they are not interoperable. Thus, 24these types are defined by Fortran, even though C may define types of the same length. 25(End of rationale.) 2627To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL*n, INTEGER*n, always create a variable whose rep-28resentation is of size **n**. These datatypes may also be used for variables declared with 29KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV 30  31 intrinsic module. Note that the MPI datatypes and the REAL*n. INTEGER*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined. 32 33 The following functions allow a user to obtain a size-specific MPI datatype for any 34intrinsic Fortran type. 35 36 MPI_SIZEOF(x, size) 37 38 IN a Fortran variable of numeric intrinsic type (choice) х 39 OUT size size of machine representation of that type (integer) 40 41 MPI_Sizeof(x, size, ierror) 42TYPE(*), DIMENSION(..) :: х 43 INTEGER, INTENT(OUT) :: size 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546MPI_SIZEOF(X, SIZE, IERROR) 47<type> X 48 INTEGER SIZE, IERROR

# Unofficial Draft for Comment Only

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C size of operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (End of advice to users.)

*Rationale.* This function is not available in other languages because it would not be useful. (End of rationale.)

## MPI_TYPE_MATCH_SIZE(typeclass, size, datatype)

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	datatype	datatype with correct type, size (handle)

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)

<pre>MPI_Type_match_size(typeclass, size, datatype, ierror)</pre>
<pre>INTEGER, INTENT(IN) :: typeclass, size</pre>
TYPE(MPI_Datatype), INTENT(OUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
 INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and MPI_TYPECLASS_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (typeclass, size).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In C, one can use the C function sizeof(), instead of MPI_SIZEOF. In addition, for variables of default kind the variable's size can be computed by a call to MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not supported by the compiler.

*Rationale.* This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (End of rationale.)

Advice to implementors. This function could be implemented as a series of tests.

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
 44
{
 switch(typeclass) {
 case MPI_TYPECLASS_REAL: switch(size) {
 case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
```

**Unofficial Draft for Comment Only** 

1

 $\mathbf{2}$ 

3

4

5

6 7

8

25

2627

28

29

30

31

32

33

34

35

36

37

38 39

40 41

4243

45

46

47

```
1
 case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
2
 default: error(...);
3
 }
4
 case MPI_TYPECLASS_INTEGER: switch(size) {
5
 case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
6
 case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
7
 default: error(...);
8
 }
9
 ... etc. ...
10
 }
11
12
 return MPI_SUCCESS;
 }
13
14
 (End of advice to implementors.)
15
16
17
 Communication With Size-specific Types
18
 The usual type matching rules apply to size-specific datatypes: a value sent with datatype
19
 MPI_{TYPE>n} can be received with this same datatype on another process. Most modern
20
 computers use 2's complement for integers and IEEE format for floating point. Thus, com-
21
 munication using these size-specific datatypes will not entail loss of precision or truncation
22
 errors.
23
24
 Advice to users. Care is required when communicating in a heterogeneous environ-
25
 ment. Consider the following code:
26
27
 real(selected_real_kind(5)) x(100)
28
 call MPI_SIZEOF(x, size, ierror)
29
 call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
30
 if (myrank .eq. 0) then
31
 ... initialize x ...
32
 call MPI_SEND(x, xtype, 100, 1, ...)
33
 else if (myrank .eq. 1) then
34
 call MPI_RECV(x, xtype, 100, 0, ...)
35
 endif
36
37
 This may not work in a heterogeneous environment if the value of size is not the
38
 same on process 1 and process 0. There should be no problem in a homogeneous
39
 environment. To communicate in a heterogeneous environment, there are at least four
40
 options. The first is to declare variables of default type and use the MPI datatypes
41
 for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
42
 is to use selected_real_kind or selected_int_kind and with the functions of the
43
 previous section. The third is to declare a variable that is known to be the same
44
 size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
45
 result in an 8-byte representation). The fourth is to carefully check representation
46
 size before communication. This may require explicit conversion to a variable of size
47
 that can be communicated and handshaking between sender and receiver to agree on
48
 a size.
```

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
 4
 call MPI_SIZEOF(x, size, ierror)
 5
 call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
 6
 7
 if (myrank .eq. 0) then
 call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
 &
 9
 MPI_MODE_CREATE+MPI_MODE_WRONLY,
 &
 10
 MPI_INFO_NULL, fh, ierror)
 11
 call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
 12
 MPI_INFO_NULL, ierror)
 13
 call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
 14
 15
 call MPI_FILE_CLOSE(fh, ierror)
 endif
 16
 17
 18
 call MPI_BARRIER(MPI_COMM_WORLD, ierror)
 19
 if (myrank .eq. 1) then
 20
 call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
 21
 &
 MPI_INFO_NULL, fh, ierror)
 22
 call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
 23
 ^{24}
 MPI_INFO_NULL, ierror)
 25
 call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
 26
 call MPI_FILE_CLOSE(fh, ierror)
 endif
 27
 28
 29
 If processes 0 and 1 are on different machines, this code may not work as expected if
 30
 the size is different on the two machines. (End of advice to users.)
 31
 32
 33
18.1.10
 Problems With Fortran Bindings for MPI
 34
This section discusses a number of problems that may arise when using MPI in a Fortran
 35
program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It
 36
is intended to clarify, not add to, this standard.
 37
 As noted in the original MPI specification, the interface violates the Fortran standard
 38
in several ways. While these may cause few problems for Fortran 77 programs, they become
 39
 40
```

In several ways. While these may cause lew 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 TS 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
 46
 types. When using the mpi_f08 module together with a compiler that supports For 47
 tran 2008 + TS 29113, this problem is resolved.

#### Unofficial Draft for Comment Only

1 2

3

41

42

43

44

- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
- 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TS 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 for more information.
- 19 6. The memory allocation routine MPI_ALLOC_MEM cannot be used from 20Fortran 77/90/95 without a language extension (for example, Cray pointers) that 21allows the allocated memory to be associated with a Fortran variable. Therefore, 22 address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, 23TYPE(C_PTR) entities were added, which allow a standard-conforming implementation 24of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has 25an additional, overloaded interface to support this language feature. The use of Cray 26pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers. 27
  - Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
    - MPI identifiers exceed 6 characters.
    - MPI identifiers may contain underscores after the first character.
    - MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
    - Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used instead.
- ⁴⁰ MPI-1 contained several routines that take address-sized information as input or return ⁴¹ address-sized information as output. In C such arguments were of type MPI_Aint and in ⁴² Fortran of type INTEGER. On machines where integers are smaller than addresses, these ⁴³ routines can lose information. In MPI-2 the use of these functions has been deprecated and ⁴⁴ they have been replaced by routines taking INTEGER arguments of KIND=MPI_ADDRESS_KIND. ⁴⁵ A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See ⁴⁶ Section 2.6 and Section 4.1.1 for more information.
- 47 Sections 18.1.11 through 18.1.19 describe several problems in detail which concern
   48 the interaction of MPI and Fortran as well as their solutions. Some of these solutions

# Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8 9

10

11

12

13

28

29

30 31

32

33 34

35

36

37

38

require special capabilities from the compilers. Major requirements are summarized in Section 18.1.7.

# 18.1.11 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 18.1.6). In C, the use of void* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the mpi_f08 module, the problem is avoided by declaring choice arguments with TYPE(*), DIMENSION(...), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
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. When using either 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)
```

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

18.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g.,

REAL a(100,100,100) CALL MPI_Send( a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)

The handling of subscript triplets depends on the value of the constant MPI_SUBARRAYS_SUPPORTED:

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 24 

1	• If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:
2	Choice buffer arguments are declared as TYPE(*), DIMENSION(). For example,
3	consider the following code fragment:
4	
5	REAL s(100), r(100)
6	CALL MPI_Isend(s(1:100:5), 3, MPI_REAL,, rq, ierror)
7	CALL MPI_Wait(rq, status, ierror)
8	CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL,, rq, ierror)
9	CALL MPI_Wait(rq, status, ierror)
10	0.122 1
11	In this case, the individual elements $s(1)$ , $s(6)$ , and $s(11)$ are sent between the start
12	of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy
13	s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code
14 15	will pass a descriptor to MPI_ISEND that allows MPI to operate directly on s(1), s(6),
16	s(11),, s(96). The called MPI_ISEND routine will take only the first three of these
10	elements due to the type signature "3, MPI_REAL".
18	
19	All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT, MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice
20	, , , , , , , , , , , , , , , , , , , ,
21	buffers are copied to a contiguous scratch buffer in the MPI runtime environment.
22	All datatype descriptions (in the example above, "3, MPI_REAL") read and store data from and to this virtual contiguous scratch buffer. Displacements in MPI de-
23	rived datatypes are relative to the beginning of this virtual contiguous scratch buffer.
24	Upon completion of a nonblocking receive operation (e.g., when MPI_WAIT on a cor-
25	responding MPI_Request returns), it is as if the received data has been copied from
26	the virtual contiguous scratch buffer back to the non-contiguous application buffer.
27	In the example above, $r(1)$ , $r(6)$ , and $r(11)$ are guaranteed to be defined with the
28	received data when MPI_WAIT returns.
29	
30	Note that the above definition does not supercede restrictions about buffers used with non-blacking energitical in Section $2.72$
31	non-blocking operations (e.g., those specified in Section $3.7.2$ ).
32	Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION()
33	arguments contains enough information that, if desired, the MPI library can make
34	a real contiguous copy of non-contiguous user buffers when the nonblocking op-
35	eration is started, and release this buffer not before the nonblocking communi-
36	cation has completed (e.g., the MPI_WAIT routine). Efficient implementations
37	may avoid such additional memory-to-memory data copying. (End of advice to
38	implementors.)
39	<b>x</b> /
40	Rationale. If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous
41	buffers are handled inside the MPI library instead of by the compiler through
42	argument association conventions. Therefore, the scope of $MPI$ library scratch
43	buffers can be from the beginning of a nonblocking operation until the completion
44	of the operation although beginning and completion are implemented in different
45	routines. (End of rationale.)
46	
47 48	• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:
48	

In this case, the use of Fortran arrays with subscript triplets as actual choice buffer arguments in any nonblocking MPI operation (which also includes persistent request, and split collectives) may cause undefined behavior. They may, however, be used in blocking MPI operations.

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.¹

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

name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )

 $\mathbf{2}$ 

 $\overline{7}$ 

 31 

 $45 \\ 46$ 

¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

	004	UNAFIER 18. LANGUAGE DINDINGS
$\frac{1}{2}$		That is, there are zero or more dimensions that are selected in full, then one dimension
3		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
4		contiguous. Examples are
5		
6 7		A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
8 9		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.
10 11 12 13 14		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
15 16 17 18 19		real :: a call user1(a,rq) call MPI_WAIT(rq,status,ierr) write (*,*) a
20 21 22 23		<pre>subroutine user1(buf,request) call MPI_IRECV(buf,,request,) end</pre>
24 25 26		If a is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter a itself.
27 28 29 30 31		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.
32 33		If a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this must be employed.
34 35 36 37 38 39 40 41		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.
42 43	18.1	13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
44 45 46		ran arrays with <b>vector</b> subscripts describe subarrays containing a possibly irregular f elements
47 48		EAL a(100) ALL MPI_Send( A((/7,9,23,81,82/)), 5, MPI_REAL,)

CHAPTER 18. LANGUAGE BINDINGS

Fortran 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.

#### 18.1.14 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. 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.

## 18.1.15 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%1, disp(4), ierr)
base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
```

Unofficial Draft for Comment Only

```
1
 disp(3) = disp(3) - base
\mathbf{2}
 disp(4) = disp(4) - base
3
4
 blocklen(1) = 1
5
 blocklen(2) = 1
6
 blocklen(3) = 1
7
 blocklen(4) = 1
8
9
 type(1) = MPI_INTEGER
10
 type(2) = MPI_REAL
11
 type(3) = MPI_DOUBLE_PRECISION
12
 type(4) = MPI_LOGICAL
13
14
 call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
15
 call MPI_TYPE_COMMIT(newtype, ierr)
16
17
 call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
18
 ! or
19
 call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
20
 ! expects that base == address(foo%i) == address(foo)
21
22
 call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
23
 call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
24
 extent = disp(2) - disp(1)
25
 1b = 0
26
 call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
27
 call MPI_TYPE_COMMIT(newarrtype, ierr)
28
29
 call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
30
 Using the derived type variable foo instead of its first basic type element foo%i may
^{31}
 be impossible if the MPI library implements choice buffer arguments through overloading
32
 instead of using TYPE(*), DIMENSION(..), or through a non-standardized extension such
33
 as !$PRAGMA IGNORE_TKR; see Section 18.1.6.
34
 To use a derived type in an array requires a correct extent of the datatype handle
35
 to take care of the alignment rules applied by the compiler. These alignment rules may
36
 imply that there are gaps between the components of a derived type, and also between the
37
 subsuguent elements of an array of a derived type. The extent of an interoperable derived
38
 type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may
39
 be different because C and Fortran may apply different alignment rules. As recommended
40
 in the advice to users in Section 4.1.6, one should add an additional fifth structure element
41
 with one numerical storage unit at the end of this structure to force in most cases that
42
 the array of structures is contiguous. Even with such an additional element, one should
43
 keep this resizing due to the special alignment rules that can be used by the compiler for
44
 structures, as also mentioned in this advice.
45
 Using the extended semantics defined in TS 29113, it is also possible to use entities
46
 or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer
```

**Unofficial Draft for Comment Only** 

arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER

47

type components may exist. In this case, the **base** address in the example must be changed to become the address of **foo** instead of **foo%i**, because the Fortran compiler may rearrange type components or add padding. Sending the structure **foo** should then also be performed by providing it (and not **foo%i**) as actual argument for MPI_Send.

## 18.1.16 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 18.1.17.
- Temporary data movement and temporary memory modifications; see Section 18.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 18.1.19.

Table 18.2 shows the only usage areas where these optimization problems may occur.

Optimization	may cause a problem in				
	fc	following usage areas			
	Nonbl.	1-sided	Split	Bottom	
Code movement	yes	yes	no	yes	
and register optimization					
Temporary data movement	yes	yes	yes	no	
Permanent data movement	yes	yes	yes	yes	

Table 18.2: Occurrence of Fortran optimization problems in several usage areas The solutions in the following sections are based on compromises:

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17 18

19

20 21

22 23

24

25 26

27 28

29 30

 31 

32 33

45 46

1 • to minimize the burden for the application programmer, e.g., as shown in Sections  $\mathbf{2}$ "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on 3 pages 669–674, 4 • to minimize the drawbacks on compiler based optimization, and 56 • to minimize the requirements defined in Section 18.1.7. 7 8 18.1.17 Problems with Code Movement and Register Optimization 9 10 Nonblocking Operations 11 If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the 12compiler will assume that it cannot be modified by a called subroutine unless it is an actual 13 argument of the call. In the most common linkage convention, the subroutine is expected 14 to save and restore certain registers. Thus, the optimizer will assume that a register which 15held a valid copy of such a variable before the call will still hold a valid copy on return. 1617**Example 18.1** Fortran 90 register optimization — extreme. 18 19Source compiled as or compiled as 20REAL :: buf, b1 21REAL :: buf, b1 REAL :: buf, b1 call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) call MPI_IRECV(buf,..req) 22 b1 = bufregister = buf 23call MPI_WAIT(req,..) call MPI_WAIT(req,..) call MPI_WAIT(req,..)  24 b1 = bufb1 = register 2526Example 18.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent 27thread modifies buf between the invocation of MPI_IRECV and the completion of MPI_WAIT. 28But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has 29returned, and may schedule the load of **buf** earlier than typed in the source. The compiler 30 has no reason to avoid using a register to hold **buf** across the call to MPI_WAIT. It also may  31 reorder the instructions as illustrated in the rightmost column. 32 33 34Example 18.2 Similar example with MPI_ISEND 35 Source compiled as with a possible MPI-internal 36 execution sequence 37 38REAL :: buf, copy REAL :: buf, copy REAL :: buf, copy 39 buf = val buf = val buf = val call MPI_ISEND(buf,..req) call MPI_ISEND(buf,...req) addr = &buf 40copy = buf copy = bufcopy= buf 41 buf = val_overwrite buf = val_overwrite 42call MPI_WAIT(req,..) call MPI_WAIT(req,..) call send(*addr) ! within 43 ! MPI_WAIT 44buf = val_overwrite 4546

⁴⁷ Due to valid compiler code movement optimizations in Example 18.2, the content of ⁴⁸ buf may already have been overwritten by the compiler when the content of buf is sent.

# Unofficial Draft for Comment Only

The code movement is permitted because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

## Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI_START and MPI_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI_START or MPI_STARTALL call, similar to the MPI_WAIT call as described in the Nonblocking Operations subsection in Section 18.1.17.

#### **One-sided** Communication

An example with instruction reordering due to register optimization can be found in Section 11.7.4.

#### MPI_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI_SEND, MPI_RECV, etc., that hides the actual variables involved in the communication. MPI_BOTTOM with an MPI_Datatype containing *absolute addresses* is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 18.3 shows what Fortran compilers are allowed to do.

In Example 18.3, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

In Example 18.4, 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.

#### Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10 11

12

13 14

15

16

17

18

19 20

21 22

23

24 25

26

27

28

29

30

 31 

32

33

34

35

36

37

38

39

40

41

42 43 44

45

46

47

```
1
 Example 18.3 Fortran 90 register optimization.
\mathbf{2}
3
 This source ...
 can be compiled as:
4
 call MPI_GET_ADDRESS(buf, bufaddr,
 call MPI_GET_ADDRESS(buf,...)
5
 ierror)
6
 call MPI_TYPE_CREATE_STRUCT(1,1,
 call MPI_TYPE_CREATE_STRUCT(...)
7
 bufaddr,
8
 MPI_REAL, type, ierror)
9
 call MPI_TYPE_COMMIT(type,ierror)
 call MPI_TYPE_COMMIT(...)
10
 val_old = buf
 register = buf
11
 val_old = register
12
 call MPI_RECV(MPI_BOTTOM,1,type,...)
 call MPI_RECV(MPI_BOTTOM,...)
13
 val_new = buf
 val_new = register
14
15
16
 Example 18.4 Similar example with MPI_SEND
17
18
 This source ...
 can be compiled as:
19
 ! buf contains val_old
 ! buf contains val_old
20
21
 buf = val_new
 call MPI_SEND(MPI_BOTTOM,1,type,...)
 call MPI_SEND(...)
22
 ! with buf as a displacement in type
 ! i.e. val_old is sent
23
 ŗ
^{24}
 ! buf=val_new is moved to here
25
26
 ! and detected as dead code
 ! and therefore removed
27
 L
28
 buf = val_overwrite
 buf = val_overwrite
29
30
^{31}
 with the send and receive buffers used in nonblocking operations, or in operations in which
32
 MPI_BOTTOM is used, or if datatype handles that combine several variables are used:
33
34
 • Use of the Fortran ASYNCHRONOUS attribute.
35
36
 • Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy
37
 routine.
38
 • Declare the buffer as a Fortran module variable or within a Fortran common block.
39
40
 • Use of the Fortran VOLATILE attribute.
41
42
 Each of these methods solves the problems of code movement and register optimization,
43
 but may incur various degrees of performance impact, and may not be usable in every
44
 application context. These methods may not be guaranteed by the Fortran standard, but
45
 they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated
46
 compiler suite according to the requirements listed in Section 18.1.7. The performance
47
 impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables
48
 or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the
```

# Unofficial Draft for Comment Only

VOLATILE attribute is expected to be high or very high. Note that there is one attribute that cannot be used for this purpose: the Fortran TARGET attribute does not solve code movement problems in MPI applications.

#### The Fortran ASYNCHRONOUS Attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi_f08 module, with allowable exceptions listed in Section 18.1.6), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if both the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 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. Section 5.4.2 of the TS 29113 specifies:

"Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent.

Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 18.5 Case (a) on page 677, 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 ⁴⁶ between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read ⁴⁷ accesses to b occur after the communication has completed. 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

1 In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to  $\mathbf{2}$ a pending communication affector while input communication (i.e., the two MPI_Irecv 3 calls) is pending. This is a contradiction to the rule that for input communication, a 4 pending communication affector shall not be referenced. The problem can be solved by using 5separate variables for the halos and the inner array, or by splitting a common array into 6 disjoint subarrays which are passed through different dummy arguments into a subroutine,  $\overline{7}$ as shown in Example 18.9.

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 18.3 and Example 18.4, 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.

13 14

21

22

34

35

36

37 38

39 40

48

8

9

10

11

12

15Calling MPI_F_SYNC_REG 16

The compiler may be prevented from moving a reference to a buffer across a call to an 17MPI subroutine by surrounding the call by calls to an external subroutine with the buffer 18 as an actual argument. The MPI library provides the MPI_F_SYNC_REG routine for this 19 purpose; see Section 18.1.8. 20

• The problems illustrated by the Examples 18.1 and 18.2 can be solved by calling MPI_F_SYNC_REG(buf) once immediately after MPI_WAIT.

Example 18.1 can be solved with	Example 18.2 can be solved with
<pre>call MPI_IRECV(buf,req)</pre>	<pre>buf = val call MPI_ISEND(buf,req) copy = buf</pre>
<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 18.3 and 18.4 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.

42	Example 18.3	Example 18.4
43	can be solved with	can be solved with
44	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
45	<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>
46	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
47		

The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store references to buf prior to MPI_RECV/MPI_SEND; the second call is needed to assure that any subsequent access to buf is not moved before MPI_RECV/SEND.

• In the example in Section 11.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 18.1, 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	15
bbbb = 777	buff = 999	16
	call MPI_F_SYNC_REG(buff)	17
call MPI_WIN_FENCE	call MPI_WIN_FENCE	18
call MPI_PUT(bbbb		19
into buff of process 2)		20
-		21
call MPI_WIN_FENCE	call MPI_WIN_FENCE	22
call MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)	23
	ccc = buff	24
		25

- The temporary memory modification problem, i.e., Example 18.6, can **not** be solved with this method.
- A User Defined Routine Instead of MPI_F_SYNC_REG

Instead of MPI_F_SYNC_REG, one can also use a user defined external subroutine, which is separately compiled:

subroutine DD(buf)
integer buf
end

Note that if the intent is declared in an explicit interface for the external subroutine, it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to MPI_RECV with MPI_BOTTOM as buffer might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document ⁴⁶ such usage in existing application programs although new applications should prefer ⁴⁷ MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to ⁴⁸

#### Unofficial Draft for Comment Only

 $41 \\ 42$ 

¹ such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because
 ² the user-written routine may not be implemented in accordance with the rules specified in

- Section 18.1.7.
- 3 4 5

6

13

14

15

16 17

18

# Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.
- 19 20 21

22

# The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution of MPI programs is discouraged.

29 30

# The Fortran TARGET Attribute

The TARGET attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the TARGET attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

*Rationale.* The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

39 40

37

38

41 42

# 18.1.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 18.5, Case (b) on page 677. Example 18.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 18.7, buf(100,100) from Example 18.6 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 18.8 shows a second possible optimization. The whole array is temporarily moved to local_buf.

When storing local_buf back to the original location buf, then this implies overwriting the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this storing back of local_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the ..._BEGIN and ..._END calls; see Section 13.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 671 of Section 18.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 18.9 and in Example 18.10.

Note also that the methods

- calling MPI_F_SYNC_REG (or such a user-defined routine),
- using module variables and COMMON blocks, and
- the TARGET attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 18.6 and 18.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the ASYNCHRONOUS attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this ⁴⁶ restriction, a new or modified asynchronous feature in the Fortran language would ⁴⁷ be necessary: an asynchronous attribute that can be used on parts of an array and ⁴⁸

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

 24 

together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (End of rationale.)

In Example 18.9 (which is a solution for the problem shown in Example 18.5 and in Example 18.10 (which is a solution for the problem shown in Example 18.8), the ar-6 ray is split into inner and halo part and both disjoint 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 10 area are strided arrays. Those can be used in non-blocking communication only with a TS 11 29113 based MPI library. 12

13 14

15

16

17

18

1920

21

22

23

 24 25

26

27

28

1

2

3

4

5

7

8

9

# 18.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. An implementation with automatic garbage collection is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with MPI_BOTTOM.
- All nonblocking MPI operations 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 18.1.7.

2930

 31 

#### Comparison with C 18.1.20

32 In C, subroutines which modify variables that are not in the argument list will not cause 33 register optimization problems. This is because taking pointers to storage objects by using 34the & operator and later referencing the objects by indirection on the pointer is an integral 35 part of the language. A C compiler understands the implications, so that the problem should 36 not occur, in general. However, some compilers do offer optional aggressive optimization 37 levels which may not be safe. Problems due to temporary memory modifications can also 38 occur in C. As above, the best advice is to avoid the problem: use different variables for 39 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 40operation is pending.

- 41
- 42
- 43
- 4445
- 46
- 47
- 48

Example 18.5 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
Example 18.6 Overlapping Communication and Computation.
USE mpi_f08
REAL :: buf(100,100)
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,...)
```

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

38

39

40

41

42

43

44

45

46

```
1
\mathbf{2}
3
4
5
6
 Example 18.7 The compiler may substitute the nested loops through loop fusion.
\overline{7}
8
 REAL :: buf(100,100), buf_1dim(10000)
9
 EQUIVALENCE (buf(1,1), buf_1dim(1))
10
 CALL MPI_Irecv(buf(1,1:100),...req,...)
11
 tmp(1:100) = buf(1,1:100)
12
 DO j=1,10000
13
 buf_1dim(h)=...
14
 END DO
15
 buf(1,1:100) = tmp(1:100)
16
 CALL MPI_Wait(req,...)
17
18
19
20
21
22
23
24
25
26
27
28
29
 Example 18.8 Another optimization is based on the usage of a separate memory storage
30
 area, e.g., in a GPU.
31
32
 REAL :: buf(100,100), local_buf(100,100)
33
 CALL MPI_Irecv(buf(1,1:100),...req,...)
34
 local_buf = buf
35
 DO j=1,100
36
 DO i=2,100
37
 local_buf(i,j)=....
38
 END DO
39
 END DO
40
 buf = local_buf ! may overwrite asynchronously received
41
 ! data in buf(1,1:100)
42
 CALL MPI_Wait(req,...)
43
44
45
46
47
48
```

**Example 18.9** 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 sent 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 18.10 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
```

# 18.2 Language Interoperability

# 18.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

**Initialization** We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.
- **Interlanguage communication** We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

# 18.2.2 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.

# 18.2.3 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

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 caller or by the processes killed. 8 9 The MPI environment is initialized in the same manner for all languages by 10 MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: 11same processes, same environmental attributes, same error handlers. Information can be added to info objects in one language and retrieved in another. 1213 Advice to users. The use of several languages in one MPI program may require the 14use of special options at compile and/or link time. (End of advice to users.) 1516Advice to implementors. Implementations may selectively link language specific MPI 17 libraries only to codes that need them, so as not to increase the size of binaries for codes 18 that use only one language. The MPI initialization code need perform initialization for 19 a language only if that language library is loaded. (End of advice to implementors.) 202118.2.4 Transfer of Handles 2223Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran  24 handles to C handles. There is no direct access to C handles in Fortran. 25The type definition MPI_Fint is provided in C for an integer of the size that matches a 26Fortran INTEGER; usually, MPI_Fint will be equivalent to int. With the Fortran mpi module 27or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in 28the following conversion functions. With the Fortran mpi_f08 module, a Fortran handle is a 29 BIND(C) derived type that contains an INTEGER component named MPI_VAL. This INTEGER 30 value can be used in the following conversion functions.  31 The following functions are provided in C to convert from a Fortran communicator 32 handle (which is an integer) to a C communicator handle, and vice versa. See also Sec-33 tion 2.6.4. 34MPI_Comm MPI_Comm_f2c(MPI_Fint comm) 35 If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a 36 valid C handle to that same communicator; if  $comm = MPI_COMM_NULL$  (Fortran value), 37 then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then 38 39 MPI_Comm_f2c returns an invalid C handle. MPI_Fint MPI_Comm_c2f(MPI_Comm comm) 4041 The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle 42to the same communicator; it maps a null handle into a null handle and an invalid handle 43 into an invalid handle. 44 Similar functions are provided for the other types of opaque objects. 45MPI_Datatype MPI_Type_f2c(MPI_Fint datatype) 46

```
47
 MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
```

1

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 20
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	
MPI_Message MPI_Message_f2c(MPI_Fint message)	
MPI_Fint MPI_Message_c2f(MPI_Message message)	
m 1_Fint m 1_Message_021(m 1_Message message)	24 25
<b>Example 18.11</b> The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function	
arguments are passed by addresses.	30
	31 32
! FORTRAN PROCEDURE SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)	
INTEGER :: DATATYPE, IERR	
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	35
RETURN	36 37
END	38
/* C wrapper */	39
	40 41
<pre>void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)</pre>	
{ MPI_Datatype datatype;	42 43
MI_Datatype datatype,	44
<pre>datatype = MPI_Type_f2c( *f_handle);</pre>	45
<pre>*ierr = (MPI_Fint)MPI_Type_commit( &amp;datatype);</pre>	46
<pre>*f_handle = MPI_Type_c2f(datatype);</pre>	47 48
return;	40

Unofficial Draft for Comment Only

1

2

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17

18

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.*)

18.2.5 Status

The following two procedures are provided in C to convert from a Fortran (with the mpi module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

 $24 \\ 25$ 

26

27

28

29

30

 31 

32

33

42

43

44

45

int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)

If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or

MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous. The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in the mpi module or mpif.h. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code. To do the conversion in the other direction, we have the following:

int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)

This call converts a C status into a Fortran status, and has a behavior similar to MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE.

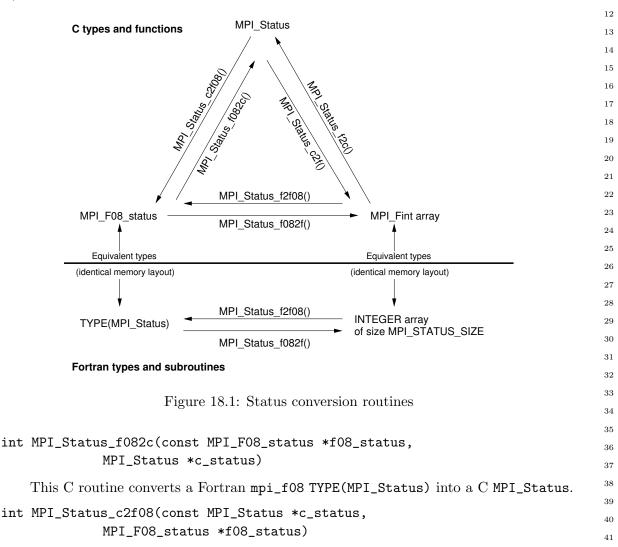
Advice to users. There exists no separate conversion function for arrays of statuses,
 since one can simply loop through the array, converting each status with the routines
 in Figure 18.1. (End of advice to users.)

# 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 18.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.



This C routine converts a C MPI_Status into a Fortran mpi_f08 TYPE(MPI_Status). ⁴² Two global variables of type MPI_F08_status*, MPI_F08_STATUS_IGNORE and ⁴³ MPI_F08_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether ⁴⁴ f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in ⁴⁵ the mpi_f08 module. These are global variables, not C constant expressions and cannot be ⁴⁶ used in places where C requires constant expressions. Their value is defined only between ⁴⁷ the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code. ⁴⁸

# Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

```
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)
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)
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
 18.2.6
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
47
 language.
```

⁴⁸ We examine below in more detail issues that arise for each type of MPI object.

# Unofficial Draft for Comment Only

#### Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see Section 18.2.9).

#### Example 18.12

```
! 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
AOBLEN(1) = 5
 21
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
 22
AOTYPE(1) = MPI_REAL
 23
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)
/* C code */
void C_ROUTINE(MPI_Fint *ftype)
{
 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
 MPI_Get_address(&count, &displs[0]);
 displs[1] = 0;
 types[0] = MPI_INT;
 types[1] = MPI_Type_f2c(*ftype);
 MPI_Type_create_struct(2, lens, displs, types, &newtype);
 MPI_Type_commit(&newtype);
 MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
 /* the message sent contains an int count of 5, followed
 */
 /* by the 5 REAL entries of the Fortran array R.
 */
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

1213

14

1516

17

18

19

20

24

25

26

2728

29

30

31

32

33

38

39

40

41

42

43

44 45

46

47

Advice to implementors. The following implementation can be used: MPI addresses, as returned by MPI_GET_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly different. A test is performed to check whether buf = MPI_BOTTOM. If true, then the base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does not have the same value in Fortran and C, then an additional test for buf = MPI_BOTTOM is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI_BOTTOM even in C, so as to distinguish it from a NULL pointer. If MPI_BOTTOM = c then one can still avoid the test buf = MPI_BOTTOM, by using the displacement from MPI_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI_GET_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

23 Callback Functions

²⁴ MPI calls may associate callback functions with MPI objects: error handlers are associ-²⁵ ated with communicators and files, attribute copy and delete functions are associated with ²⁶ attribute keys, reduce operations are associated with operation objects, etc. In a multilan-²⁷ guage environment, a function passed in an MPI call in one language may be invoked by an ²⁹ MPI call in another language. MPI implementations must make sure that such invocation ²⁰ 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 291. (*End of advice to users.*)

## Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length argument list.
 It might be useful to provide to the handler information on the language environment
 where the error occurred. (*End of advice to implementors.*)

## 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

36

37

38

39

40

41

42

43 44

#### **Reduce Operations**

All predefined named and unnamed datatypes as listed in Section 5.9.2 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.*)

#### 18.2.7 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 defines attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 6.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer-valued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if 

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

 $\overline{7}$ 

1 deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and  $\mathbf{2}$ MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr. 3 Example 18.13 4 A. Setting an attribute value in C 56 int set_val = 3; 7 struct foo set_struct; 8 9 /* Set a value that is a pointer to an int */ 10 11 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val); 12/* Set a value that is a pointer to a struct */ 13 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct); 14 /* Set an integer value */ 15 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17); 16 17B. Reading the attribute value in C 18 19 int flag, *get_val; 20struct foo *get_struct; 2122/* Upon successful return, get_val == &set_val 23(and therefore *get_val == 3) */  24 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag); 25/* Upon successful return, get_struct == &set_struct */ 26MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag); 27/* Upon successful return, get_val == (void*) 17 */ 28 i.e., (MPI_Aint) get_val == 17 */ /* 29MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag); 30 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 3132 LOGICAL FLAG 33 INTEGER IERR, GET_VAL, GET_STRUCT 34 35 ! Upon successful return, GET_VAL == &set_val, possibly truncated 36 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR) 37 ! Upon successful return, GET_STRUCT == &set_struct, possibly truncated 38 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR) 39 ! Upon successful return, GET_VAL == 17 40CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR) 41 42D. Reading the attribute value with Fortran MPI-2 calls 43 44 4546 47 48

```
1
LOGICAL FLAG
 \mathbf{2}
INTEGER IERR
 3
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
 4
! Upon successful return, GET_VAL == &set_val
 5
 6
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
 7
! Upon successful return, GET_STRUCT == &set_struct
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
 8
! Upon successful return, GET_VAL == 17
 9
 10
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
 11
 12
Example 18.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
 13
 14
INTEGER IERR, VAL
 15
VAL = 7
 16
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
 17
 18
 B. Reading the attribute value in C
 19
 20
int flag;
 21
int *value;
 22
 23
/* Upon successful return, value points to internal MPI storage and
 ^{24}
 *value == (int) 7 */
 25
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
 26
 27
 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
 28
 29
LOGICAL FLAG
 30
INTEGER IERR, VALUE
 31
 32
! Upon successful return, VALUE == 7
 33
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
 34
 D. Reading the attribute value with Fortran MPI-2 calls
 35
 36
LOGICAL FLAG
 37
INTEGER IERR
 38
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
 39
 40
! Upon successful return, VALUE == 7 (sign extended)
 41
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
 42
 43
 44
Example 18.15 A. Setting an attribute value via a Fortran MPI-2 call
 45
 46
 47
```

```
1
 INTEGER IERR
\mathbf{2}
 INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
3
 INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
4
 VALUE1 = 42
\mathbf{5}
 VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
6
7
 CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
8
 CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
9
 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<sup>40</sup>, or 0 if truncation
29
 ! needed (i.e., the least significant part of the attribute word)
30
 CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
^{31}
32
 D. Reading the attribute value with Fortran MPI-2 calls
33
34
 LOGICAL FLAG
35
 INTEGER IERR
36
 INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
37
38
 ! Upon successful return, VALUE1 == 42
39
 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
40
 ! Upon successful return, VALUE2 == 2^40
41
 CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
42
43
 The predefined MPI attributes can be integer valued or address-valued. Predefined
44
 integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to
45
 the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran,
46
 MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return
47
 in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD,
48
```

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.)

## 18.2.8 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.

## 18.2.9 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 18.2.4. 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 

2

3

4

5 6

7

8

9

10

11

12

13

### (End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the Datatypes subsection in Section 18.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (End of rationale.)

#### 14 15 16

17

## 18.2.10 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.

24

Example 18.16 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.

### 

## 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)	
(command on none 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	MPI_ERR_BASE
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_NOKEY
18	MPI_ERR_INFO_VALUE
19	MPI_ERR_INFO
20	MPI_ERR_IO
21	MPI_ERR_KEYVAL
22	MPI_ERR_LOCKTYPE
23	
24	MPI_ERR_NAME
25	MPI_ERR_NO_MEM
	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_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	3
MPI_T_ERR_C	CANNOT_INIT	4
MPI_T_ERR_N	IOT_INITIALIZED	5
MPI_T_ERR_N	IEMORY	6
MPI_T_ERR_I	NVALID	7
MPI_T_ERR_I	NVALID_INDEX	8
MPI_T_ERR_I	$NVALID_ITEM$ (deprecated as of $MPI$	-3.2) 9
MPI_T_ERR_I	NVALID_SESSION	10
MPI_T_ERR_I	NVALID_HANDLE	11
MPI_T_ERR_I	NVALID_NAME	12
MPI_T_ERR_C	OUT_OF_HANDLES	13
MPI_T_ERR_C	OUT_OF_SESSIONS	14
MPI_T_ERR_C	VAR_SET_NOT_NOW	15
	VAR_SET_NEVER	16
	vVAR_NO_WRITE	17
	vAR_NO_STARTSTOP	18
	VAR_NO_ATOMIC	19
MPI_ERR_LAS		20
		21
E	Buffer Address Constants	22
C type: void * const		23
Fortran type: (predefine	ed memory location) ^{$1$}	24
MPI_BOTTOM		25
MPI_IN_PLACE		26
$^{-1}$ Note that in Fortra:	n these constants are not usable for	initialization 27
expressions or assign	ament. See Section 2.5.4.	28
		29
	Assorted Constants	30
C ty	pe: const int (or unnamed enum)	31
Fort	an 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
	ROOT	41
		42
Ne	o Process Message Handle	43
C type:	MPI_Message	- 44
• -	type: INTEGER or TYPE(MPI_Message)	45
	SSAGE_NO_PROC	- 46
		- 47
		48

	Fortran Support Method Specific Constan
	Fortran type: LOGICAL
	MPI_SUBARRAYS_SUPPORTED (Fortran only)
	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran of
	Status size and reserved index values (Fortran
_	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 char	
	(treated as integral value)	
MPI_UNSIGNED_CHAR	unsigned char	
	(treated as integral value)	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long	
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	For	rtran types
2	C type: MPI_Datatype		
3	Fortran type: INTEGER		
4	or TYPE(MPI_Datatype)		
5	MPI_INTEGER	INT	EGER
6	MPI_REAL	REA	L
7	MPI_DOUBLE_PRECISION	DOU	BLE PRECISION
8	MPI_COMPLEX	COM	IPLEX
9	MPI_LOGICAL	LOG	ICAL
10	MPI_CHARACTER	CHA	RACTER(1)
11	MPI_AINT	INT	EGER (KIND=MPI_ADDRESS_KIND)
12	MPI_COUNT	INT	EGER (KIND=MPI_COUNT_KIND)
13	MPI_OFFSET		EGER (KIND=MPI_OFFSET_KIND)
14	MPI_BYTE		y Fortran type)
15	 MPI_PACKED		y Fortran type)
16			
17	Named Predefined Datatype	$\mathbf{es}^1$	C++ types
18	C type: MPI_Datatype		
19	Fortran type: INTEGER		
20	or TYPE(MPI_Datatype)		
21	MPI_CXX_BOOL		bool
22	MPI_CXX_FLOAT_COMPLEX		<pre>std::complex<float></float></pre>
23	MPI_CXX_DOUBLE_COMPLEX		<pre>std::complex<double></double></pre>
24	MPI_CXX_LONG_DOUBLE_COMP	LEX	<pre>std::complex<long double=""></long></pre>
25	$^{-1}$ If an accompanying C++ comp	piler	is missing, then the
26	MPI datatypes in this table are	e not	defined.
27			
28	Optional datatypes (I	Fortr	an) Fortran types
29	$\mathrm{C} \ \mathrm{type}$ : MPI_Datatype		
30	Fortran type: INTEGER		
31	or TYPE(MPI_Datatype)		
32	MPI_DOUBLE_COMPLEX		DOUBLE COMPLEX
33	MPI_INTEGER1		INTEGER*1
34	MPI_INTEGER2		INTEGER*2
35	MPI_INTEGER4		INTEGER*4
36	MPI_INTEGER8		INTEGER*8
37	MPI_INTEGER16		INTEGER*16
38	MPI_REAL2		REAL*2
39	MPI_REAL4		REAL*4
40	MPI_REAL8		REAL*8
41	MPI_REAL16		REAL*16
42	MPI_COMPLEX4		COMPLEX*4
43	MPI_COMPLEX8		COMPLEX*8
44	MPI_COMPLEX16		COMPLEX*16
45	MPI_COMPLEX32		COMPLEX*32
46			
47			
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
Datatypes for reduction functions (Fortran)	10
C type: MPI_Datatype	12
Fortran type: INTEGER or TYPE(MPI_Datatype)	13
MPI_2REAL	14
MPI_2DOUBLE_PRECISION	15
MPI_2INTEGER	16
	17
Reserved communicators	18
C type: MPI_Comm	19
Fortran type: INTEGER or TYPE(MPI_Comm)	20
MPI_COMM_WORLD	21
MPI_COMM_SELF	22
	23
Communicator split type constants	24
C type: const int (or unnamed enum)	25
Fortran type: INTEGER	26
MPI_COMM_TYPE_SHARED	27
	28
Results of communicator and group comparisons	29
C type: const int (or unnamed enum)	30
Fortran type: INTEGER	31
MPI_IDENT	32
MPI_CONGRUENT	33
MPI_SIMILAR	34
MPI_UNEQUAL	35
Environmental inquiry info key	37
C type: MPI_Info	38
Fortran type: INTEGER or TYPE(MPI_Info)	39
MPI_INFO_ENV	40
	41
Environmental inquiry keys	42
C type: const int (or unnamed enum)	43
Fortran type: INTEGER	44
MPI_TAG_UB	45
MPI_IO	46
MPI_HOST	47
MPI_WTIME_IS_GLOBAL	48

**Unofficial Draft for Comment Only** 

Collective Operations
C type: MPI_Op
Fortran type: INTEGER or TYPE(MPI_Op)
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_MAXLOC
MPI_MINLOC
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR
MPI_REPLACE
MPI_NO_OP
Null Handles
C/Fortran name
C type / Fortran type
· / ·
MPI_GROUP_NULL
MPI_Group / INTEGER or TYPE(MPI_Group)
MPI_COMM_NULL
MPI_Comm / INTEGER or TYPE(MPI_Comm)
MPI_DATATYPE_NULL
$ t MPI_Datatype \ / \  ext{INTEGER} \  ext{or TYPE(MPI_Datatype)}$
MPI_REQUEST_NULL
<pre>MPI_Request / INTEGER or TYPE(MPI_Request) MPI_OP_NULL</pre>
MPI_Op / INTEGER or TYPE(MPI_Op)
MPI_ERRHANDLER_NULL
MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)
MPI_FILE_NULL
MPI_File / INTEGER or TYPE(MPI_File)
MPI_INFO_NULL
<pre>MPI_Info / INTEGER or TYPE(MPI_Info)</pre>
MPI_WIN_NULL
MPI_Win / INTEGER or TYPE(MPI_Win)
MPI_MESSAGE_NULL
MPI_Message / INTEGER or TYPE(MPI_Message)
Franty group
Empty group
C type: MPI_Group
Fortran type: INTEGER or TYPE(MPI_Group)
MPI_GROUP_EMPTY

	Topologies
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_GRAPH
	MPI_CART
	MPI_DIST_GRAPH
	Predefined functions
C/Fortran name	
C type	
/ Fortran type with mp:	i module / Fortran type with mpi_f08 module
MPI_COMM_NULL_COP	PY_FN
MPI_Comm_copy_attr_	function
/ COMM_COPY_ATTR_FUN	$(\texttt{TTION} / \texttt{PROCEDURE}(\texttt{MPI_Comm_copy_attr_function})^1)$
MPI_COMM_DUP_FN	
MPI_Comm_copy_attr_i	
/ COMM_COPY_ATTR_FUN	,
MPI_COMM_NULL_DEL	
MPI_Comm_delete_att	
/ COMM_DELETE_ATTR_F	
MPI_Win_copy_attr_fi	
/ WIN_COPY_ATTR_FUNC	
MPI_WIN_DUP_FN	
MPI_Win_copy_attr_f	inction
/ WIN_COPY_ATTR_FUNC	_
MPI_WIN_NULL_DELET	TE_FN
MPI_Win_delete_attr	_function
/ WIN_DELETE_ATTR_FU	$egin{array}{llllllllllllllllllllllllllllllllllll$
MPI_TYPE_NULL_COP	Y_FN
MPI_Type_copy_attr_	function
/ TYPE_COPY_ATTR_FUN	$(\texttt{TTION} / \texttt{PROCEDURE}(\texttt{MPI_Type_copy_attr_function})^1)$
MPI_TYPE_DUP_FN	
MPI_Type_copy_attr_t	
/ TYPE_COPY_ATTR_FUN	
MPI_TYPE_NULL_DELE	
MPI_Type_delete_att	
/ TYPE_DELETE_ATTR_F MPI_CONVERSION_FN	
MPI_Datarep_convers:	
/ DATAREP_CONVERSION	
	ementors (on page 290) and advice to users (on page 291)
	tran functions MPI_COMM_NULL_COPY_FN, in
Section $6.7.2$ .	

1	Deprecated predefined functions
2	C/Fortran name
3	C type / Fortran type with mpi module
4	MPI_NULL_COPY_FN
5	MPI_Copy_function / COPY_FUNCTION
6	MPI_DUP_FN
7	MPI_Copy_function / COPY_FUNCTION
8	MPI_NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUNCTION
10	
11	Predefined Attribute Keys
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_APPNUM
15	MPI_LASTUSEDCODE
16	MPI_UNIVERSE_SIZE
17	MPI_WIN_BASE
18	MPI_WIN_DISP_UNIT
19	MPI_WIN_SIZE
20	MPI_WIN_CREATE_FLAVOR
21	MPI_WIN_MODEL
22	
23	<b>MPI</b> Window Create Flavors
24	C type: const int (or unnamed enum)
25	Fortran type: INTEGER
26	MPI_WIN_FLAVOR_CREATE
27	MPI_WIN_FLAVOR_ALLOCATE
28	MPI_WIN_FLAVOR_DYNAMIC
29	MPI_WIN_FLAVOR_SHARED
30	WFI_WIN_FLAVOR_SHARED
31	MPI Window Models
32	C type: const int (or unnamed enum)
33	
34	Fortran type: INTEGER MPI_WIN_SEPARATE
35	
36	MPI_WIN_UNIFIED
37	
38	
39	
40	
40	
42	
43	
44	
45	
46	
47	
48	

Mode Constants	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
MPI_MODE_APPEND	4
MPI_MODE_CREATE	5
MPI_MODE_DELETE_ON_CLOSE	6
MPI_MODE_EXCL	7
MPI_MODE_NOCHECK	8
MPI_MODE_NOPRECEDE	9
MPI_MODE_NOPUT	10
MPI_MODE_NOSTORE	11
MPI_MODE_NOSUCCEED	12
MPI_MODE_RDONLY	13
MPI_MODE_RDWR	14
MPI_MODE_SEQUENTIAL	15
MPI_MODE_UNIQUE_OPEN	16
MPI_MODE_WRONLY	17
	18
Datatype Decoding Constants	19
C type: const int (or unnamed enum)	20
Fortran type: INTEGER	21
MPI_COMBINER_CONTIGUOUS	22
MPI_COMBINER_DARRAY	23
MPI_COMBINER_DUP	24
MPI_COMBINER_F90_COMPLEX	25
MPI_COMBINER_F90_INTEGER	26
MPI_COMBINER_F90_REAL	27
MPI_COMBINER_HINDEXED	28
MPI_COMBINER_HVECTOR	29
MPI_COMBINER_INDEXED_BLOCK	30
MPI_COMBINER_HINDEXED_BLOCK	31
MPI_COMBINER_INDEXED	32
MPI_COMBINER_NAMED	33
MPI_COMBINER_RESIZED	34
MPI_COMBINER_STRUCT	35
MPI_COMBINER_SUBARRAY	36
MPI_COMBINER_VECTOR	37
Threads Constants	38 39
	40
C type: const int (or unnamed enum) Fortran type: INTEGER	41
MPI_THREAD_FUNNELED	42
MPI_THREAD_FONNELED MPI_THREAD_MULTIPLE	43
MPI_THREAD_MOLTIPLE MPI_THREAD_SERIALIZED	44
MPI_THREAD_SERIALIZED MPI_THREAD_SINGLE	45
	46
	47

C type: const MP1_Offset (or unnamed enum) Fortran type: INTEGER (KIND=MP1_OFFSET_KIND) MP1_DISPLACEMENT_CURRENT File Operation Constants, Part 2 C type: const int (or unnamed enum) Fortran type: INTEGER MP1_DISTRIBUTE_CYCLIC MP1_DISTRIBUTE_CYCLIC MP1_DISTRIBUTE_NONE MP1_ORDER_C MP1_ORDER_CORTAN MP1_SEEK_CUR MP1_SEEK_CUR MP1_SEEK_SET F90 Datatype Matching Constants C type: const int (or unnamed enum) Fortran type: INTEGER MP1_TYPECLASS_COMPLEX MP1_TYPECLASS_COMPLEX MP1_TYPECLASS_COMPLEX MP1_TYPECLASS_COMPLEX MP1_TYPECLASS_REAL C C/Fortran name C type / Fortran type ¹ MP1_AGVS_NULL char*** / 2-dim. array of CHARACTER*(*) MP1_AGVS_NULL char*** / array of CHARACTER*(*) MP1_Status* / INTEGER, DIMENSION(MP1_STATUS_SIZE,*) or TYPE(MP1_Status), DIMENSION(MP1_STATUS_SIZE,*) or TYPE(MP1_Status), DIMENSION(MP1_STATUS_SIZE) or TYPE(MP1_Status) MP1_Status* / INTEGER, DIMENSION(MP1_STATUS_SIZE) or TYPE(MP1_Status) MP1_Status* / INTEGER array MP1_WEIGHTED int* / INTEGER array MP1_WEIGHTED int* / INTEGER array NP1_WEIGHTED int* / INTEGER array NP1_WEIGHTED int* / INTEGER array	1	File Operation Constants, Part 1
Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)         MPI_DISFLACEMENT_CURRENT         File Operation Constants, Part 2         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_OPIT_DARG         MPI_ORDER_C         MPI_SEEK_CUR         MPI_SEEK_ET         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_SEEK_SET         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_STATUSE_IGNORE         int* / INTEGER array         MPI_STATUSE_IGNORE         MPI_STATUSE_IGNORE         MPI_STATUSE_IGNORE         MPI_STATUSE_IGNORE	2	
MPI_DISPLACEMENT_CURRENT         File Operation Constants, Part 2         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_DISTRIBUTE_BCOCK         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_OPLIC         MPI_DISTRIBUTE_OPLIC         MPI_ORDER_C         MPI_ORDER_CORTRAN         MPI_SEEK_CUR         MPI_SEEK_SET         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_SEEK_SET         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_SEEK_SET         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_AGVS_NULL         chara** / ar	3	
File Operation Constants, Part 2         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPL_DISTRIBUTE_BLOCK         MPL_DISTRIBUTE_OFLT_DARG         MPL_ORDER_C         MPL_ORDER_CORTRAN         MPL_SEEK_CUR         MPL_SEEK_CUR         MPL_SEEK_CUR         MPL_SEEK_SET         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPL_TYPECLASS_COMPLEX         MPL_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPL_STATUSE_IGNORE         MPL_STATUSE_S_IGNORE	4	* -
C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_OFLT_DARG         MPI_ORDER_C         MPI_ORDER_C         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_SET         B         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         MPI_ARGVS_NULL         char*** / aring of CHARACTER*(*)         MPI_ARGVS_NULL         char*** / aring of CHARACTER*(*)         MPI_STATUSE_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_STATUS_GRORE         MPI_STATUS_EMPTY         int* / INTEGER array         MPL_WEIGHTE         MPI_WEIGHTED         int* / INTEGER array         MPI_WEIGHTED         int* / INTEGER array         MPI_WEIGHTE         MPI_WEIGH	5	
C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_OFLT_DARG         MPI_ORDER_C         MPI_ORDER_C         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_SET         B         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         MPI_ARGVS_NULL         char*** / aring of CHARACTER*(*)         MPI_ARGVS_NULL         char*** / aring of CHARACTER*(*)         MPI_STATUSE_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_STATUS_GRORE         MPI_STATUS_EMPTY         int* / INTEGER array         MPL_WEIGHTE         MPI_WEIGHTED         int* / INTEGER array         MPI_WEIGHTED         int* / INTEGER array         MPI_WEIGHTE         MPI_WEIGH	6	File Operation Constants, Part 2
Fortran type: INTEGER         MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_OPLT_DARG         MPI_ORDER_C         MPI_ORDER_C         MPI_SEEK_CUR         MPI_SEEK_CUR         MPI_SEEK_SET         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         C/Fortran name         C/fortran name         C type / Fortran type!         MPI_ARGVS_NULL         char** / 2-dim. array of CHARACTER*(*)         MPI_ESTATUSES_IGNORE         int* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_STATUSES_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_STATUS_ENPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         ' Note that in Fortran these constants are not usable for init	7	,
MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_DARG         MPI_DISTRIBUTE_NONE         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_END         MPI_SEEK_SET         Gottatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char** / 2-dim. array of CHARACTER*(*)         MPI_ARGVS_NULL         char** / 2-dim. array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         ' Note that in Fortran	8	
MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_DFLT_DARG         MPI_DISTRIBUTE_NONE         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_END         MPI_SEEK_SET         MPI_SEEK_SET         MPI_SEEK_SET         MPI_SEEK_SET         MPI_SEEK_SET         MPI_SEEK_SET         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_NULL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type ¹ MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / Array of CHARACTER*(*)         MPI_STATUS_GIONCE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_UNWEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY	9	· -
MPI_DISTRIBUTE_DFLT_DARG         MPI_DISTRIBUTE_NONE         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         MPI_SEEK_SET         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type¹         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_STATUSE, IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUS_SIGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_UWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran these c	10	
MPI_DISTRIBUTE_NONE         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_END         MPI_SEEK_SET         Potrantype Matching Constants         C type: const int (or unnamed enum)         Potran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_NULE         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type ¹ MPI_ARGV_NULL         char*** / aray of CHARACTER*(*)         MPI_ARGV_NULL         char** / aray of CHARACTER*(*)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY	11	
MPI_ORDER_C         MPI_SEK_CUR         MPI_SEK_END         MPI_SEK_END         MPI_SEK_END         MPI_SEK_SET         B         PO Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         C/Fortran name         C type / Fortran type ¹ MPI_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_ERRCODES_IGNORE         int* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(*)         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_SIGNORE         MPI_STATUS_IGNORE         MPI_STATUS_SIGNORE         MPI_STATUS_SIGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_REC, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_STATUS_IGNORE         MPI_WEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array	12	
MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         MPI_SEEK_SET         Sector         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         C type / Fortran name         C type / Fortran type ¹ MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran these constants are not usable for initialization         expressions or assignment. See Section 2.5.4.	13	
MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         MPI_SEEK_SET         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         C type / Fortran type ¹ MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran these constants are not usable for initialization         expressions or assignment. See Section 2.5.4.	14	
MPI_SEEK_END         MPI_SEEK_SET         Proprint         MPI_SEEK_SET         Proprint	15	
Impl_SEEK_SET         Impl_SEEK_SET         Impl_SEEK_SET         Impl_Statuspe Matching Constants         Impl_ARGVS_NULL         Impl_ARGVS_NULL         Impl_ARGVS_NULL         Impl_ARGV_NULL         Impl_ARGV_NULL         Impl_ARGV_NULL         Impl_ARGV_NULL         Impl_Statuspe Matching Constants         Impl_MPLStatuspe Matching Constants         Impl_MPLWEIGHTED         Impl_MPLWEIGHTS_EMPTY         Impl_MPLWEIGER array         Impl_MPLWEIGER array         Impl_MPLWEIGER array </td <td>16</td> <td></td>	16	
F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type!         MPI_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status)         MPI_STATUS_EGNORE         MPI_STATUS_IGNORE         MPI_STATUS_GRORE         MPI_STATUS_IGNORE         MPI_STATUS_GRORE         MPI_STATUS_GRORE         MPI_STATUS_FORORE         MPI_STATUS_GRORE         MPI_STATUS_GRORE         MPI_STATUS_GRORE         MPI_STATUS_GRORE         MPI_STATUS_GRORE         MPI_STATUS_GRORE         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran these	17	
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       Constants Specifying Empty or Ignored Input         26       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type!         29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSE_IGNORE         36       MPI_STATUSE_IGNORE         37       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         38       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         31       MPI_WEIGHTED_         32       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         34       MPI_UNWEIGHTED         35       MPI_UNWEIGHTED         36       MPI_WEIGHTS_EMPTY         37       in	18	
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       Constants Specifying Empty or Ignored Input         26       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type!         29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSE_IGNORE         36       MPI_STATUSE_IGNORE         37       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         38       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         31       MPI_WEIGHTED_         32       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         34       MPI_UNWEIGHTED         35       MPI_UNWEIGHTED         36       MPI_WEIGHTS_EMPTY         37       in	19	F90 Datatype Matching Constants
Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array	20	
MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(*)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_NetIGHTS_EMPTY         int* / INTEGER array	21	
23       MPI_TYPECLASS_INTEGER         24       MPI_TYPECLASS_REAL         25       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type1         29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         4       int* / INTEGER array         35       MPI_STATUSES_IGNORE         4       int* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         4       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         40       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_UNWEIGHTED         44       int* / INTEGER array         45       1 Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	22	V_
P24       MPI_TYPECLASS_REAL         P25       Constants Specifying Empty or Ignored Input         P27       C/Fortran name         P38       C type / Fortran type1         P39       MPI_ARGVS_NULL         P41       char*** / 2-dim. array of CHARACTER*(*)         P41       MPI_ARGV_NULL         P42       char** / array of CHARACTER*(*)         P43       MPI_ARGV_NULL         P44       char** / array of CHARACTER*(*)         P45       MPI_STATUSES_IGNORE         P44       int* / INTEGER array         P45       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         P45       MPI_STATUS_IGNORE         P46       MPI_STATUS_IGNORE         P47       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         P48       MPI_STATUS_IGNORE         P49       MPI_STATUS_SIZE)         P40       or TYPE(MPI_Status)         P41       MPI_UNWEIGHTED         P43       int* / INTEGER array         P44       int* / INTEGER array         P45       Note that in Fortran these constants are not usable for initialization         P44       expressions or assignment. See Section 2.5.4.	23	
25       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type1         29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         41       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         42       int* / INTEGER array         43       MPI_UNWEIGHTED         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	24	
27       C/Fortran name         28       C type / Fortran type ¹ 29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	25	
28       C type / Fortran type ¹ 29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	26	Constants Specifying Empty or Ignored Input
29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45       1 Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	27	C/Fortran name
MPI_ARGVS_NULL         and the second secon	28	C type / Fortran type ¹
31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	29	MPI_ARGVS_NULL
32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	30	<pre>char*** / 2-dim. array of CHARACTER*(*)</pre>
33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	31	MPI_ARGV_NULL
<ul> <li>int* / INTEGER array</li> <li>MPI_STATUSES_IGNORE</li> <li>MPI_STATUSES_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	32	<pre>char** / array of CHARACTER*(*)</pre>
<ul> <li>MPI_STATUSES_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	33	MPI_ERRCODES_IGNORE
36       MPI_STATUSES_IGNORE         36       MPI_STATUSES_IGNORE         37       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 ¹ Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	34	<pre>int* / INTEGER array</pre>
<ul> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATOS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>¹ Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	35	MPI_STATUSES_IGNORE
<ul> <li>MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	36	<pre>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</pre>
<ul> <li>MPI_STATUS_IGNORE</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>¹ Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	37	or TYPE(MPI_Status), DIMENSION(*)
<ul> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATOS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED         <ul> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>¹ Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul> </li> </ul>	38	MPI_STATUS_IGNORE
<ul> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>¹ Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	39	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
<ul> <li>MPI_ONWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>¹ Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	40	
<ul> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>¹ Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	41	
<ul> <li>⁴³ MPI_WEIGHTS_EMPTY</li> <li>⁴⁴ int* / INTEGER array</li> <li>⁴⁵ ¹ Note that in Fortran these constants are not usable for initialization ⁴⁶ expressions or assignment. See Section 2.5.4.</li> </ul>	42	
<ul> <li>int* / INTEGER array</li> <li>⁴⁵</li> <li>¹ Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	43	
<ul> <li>⁴⁵</li> <li>¹ Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	44	
46 expressions or assignment. See Section 2.5.4.	45	
47	46	
48	47	
	48	

C Constants Specify	ing Ignored Input (no Fortran)
ype: MPI_Fint*	equivalent to Fortran
PI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.
PI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h
type: MPI_F08_status*	equivalent to Fortran
PI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08
PI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08
C preprocessor Con	stants and Fortran Parameters
	nacro that expands to an int value
Fortran type: INTEGER	r in the second s
MPI_SUBVERSION	
MPI_VERSION	
Null handles used in t	he MPI tool information interface
MPI_T_ENUM_NULL	
MPI_T_enum	
MPI_T_CVAR_HANDLE_NU	LL
MPI_T_cvar_handle	
MPI_T_PVAR_HANDLE_NU	LL
MPI_T_pvar_handle	
MPI_T_PVAR_SESSION_NU	LL
MPI_T_pvar_session	
	e MPI tool information interface
C type: const int (or unn	,
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_MPID	
MPI_T_VERBOSITY_MPID	
MPI_T_VERBOSITY_MPID	EV_ALL

**Unofficial Draft for Comment Only** 

Constants to identify associations of variable
in the MPI tool information interface
C type: const int (or unnamed enum)
MPI_T_BIND_NO_OBJECT
MPI_T_BIND_MPI_COMM
MPI_T_BIND_MPI_DATATYPE
MPI_T_BIND_MPI_ERRHANDLER
MPI_T_BIND_MPI_FILE
MPI_T_BIND_MPI_GROUP
MPI_T_BIND_MPI_OP
MPI_T_BIND_MPI_REQUEST
MPI_T_BIND_MPI_WIN
MPI_T_BIND_MPI_MESSAGE
MPI_T_BIND_MPI_INFO
Constants describing the scope of a control varia
in the MPI tool information interface
C type: const int (or unnamed enum)
MPI_T_SCOPE_CONSTANT
MPI_T_SCOPE_READONLY
MPI_T_SCOPE_LOCAL
MPI_T_SCOPE_GROUP
MPI_T_SCOPE_GROUP_EQ
MPI_T_SCOPE_ALL
MPI_T_SCOPE_ALL_EQ
Additional constants used
by the MPI tool information interface
C type: MPI_T_pvar_handle
MPI_T_PVAR_ALL_HANDLES
Denformance uprichlag alagges used by the
Performance variables classes used by the MPI tool information interface
C type: const int (or unnamed enum)
MPI_T_PVAR_CLASS_STATE MPI_T_PVAR_CLASS_LEVEL
MPLI PVAR CLASS LEVEL
MPI_T_PVAR_CLASS_SIZE
MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE
MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK
MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK MPI_T_PVAR_CLASS_LOWWATERMARK
MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK MPI_T_PVAR_CLASS_LOWWATERMARK MPI_T_PVAR_CLASS_COUNTER
MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK MPI_T_PVAR_CLASS_LOWWATERMARK MPI_T_PVAR_CLASS_COUNTER MPI_T_PVAR_CLASS_AGGREGATE
MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK MPI_T_PVAR_CLASS_LOWWATERMARK MPI_T_PVAR_CLASS_COUNTER

## A.1.2 Types

47

⁴⁸ The following are defined C type definitions, included in the file mpi.h.

/* C opaque types */	1
MPI_Aint	2
MPI_Count	3
MPI_Fint	4
MPI_Offset	5
MPI_Status	6
MPI_F08_status	7
	8
/* C handles to assorted structures */	9
MPI_Comm	10
MPI_Datatype	11
MPI_Errhandler	12
MPI_File	13
MPI_Group	14
MPI_Info	15
	16
MPI_Message	17
MPI_Op	18
MPI_Request	19
MPI_Win	20
	20
<pre>/* Types for the MPI_T interface */</pre>	21
MPI_T_enum	22
MPI_T_cvar_handle	
MPI_T_pvar_handle	24
MPI_T_pvar_session	25
	26
	27
The following are defined Fortran type definitions, included in the mpi_f08 and mpi	28
modules.	29
! Fortran opaque types in the mpi_f08 and mpi modules	30
TYPE(MPI_Status)	31
	32
! Fortran handles in the mpi_f08 and mpi modules	33
TYPE(MPI_Comm)	34
TYPE(MPI_Datatype)	35
TYPE(MPI_Errhandler)	36
TYPE(MPI_File)	37
TYPE(MPI_Group)	38
TYPE(MPI_Info)	39
TYPE(MFI_INIO) TYPE(MPI_Message)	40
•	41
TYPE(MPI_Op)	42
TYPE(MPI_Request)	43
TYPE(MPI_Win)	44
	45
	46
	47
	48

1 A.1.3 Prototype Definitions  $\mathbf{2}$ C Bindings 3 4 The following are defined C typedefs for user-defined functions, also included in the file 5mpi.h. 6  $\overline{7}$ /* prototypes for user-defined functions */ typedef void MPI_User_function(void *invec, void *inoutvec, int *len, 8 9 MPI_Datatype *datatype); 10 11 typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, 12void *attribute_val_out, int *flag); 13 14typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval, void *attribute_val, void *extra_state); 151617typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval, 18 void *extra_state, void *attribute_val_in, 19 void *attribute_val_out, int *flag); typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval, 20void *attribute_val, void *extra_state); 212223typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,  24 int type_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag); 2526typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype, int type_keyval, void *attribute_val, void *extra_state); 2728typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...); 29typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...); 30  31 typedef void MPI_File_errhandler_function(MPI_File *, int *, ...); 3233 typedef int MPI_Grequest_query_function(void *extra_state, 34 MPI_Status *status); typedef int MPI_Grequest_free_function(void *extra_state); 35 typedef int MPI_Grequest_cancel_function(void *extra_state, int complete); 36 37 typedef int MPI_Datarep_extent_function(MPI_Datatype datatype, 3839 MPI_Aint *file_extent, void *extra_state); typedef int MPI_Datarep_conversion_function(void *userbuf, 4041 MPI_Datatype datatype, int count, void *filebuf, 42MPI_Offset position, void *extra_state); 43 44Fortran 2008 Bindings with the mpi_f08 Module 45The callback prototypes when using the Fortran mpi_f08 module are shown below: 46The user-function argument to MPI_Op_create should be declared according to: 47ABSTRACT INTERFACE 48

```
1
 SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
 2
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
 3
 TYPE(C_PTR), VALUE :: invec, inoutvec
 INTEGER :: len
 4
 TYPE(MPI_Datatype) :: datatype
 5
 6
 The copy and delete function arguments to MPI_Comm_create_keyval should be de-
 7
clared according to:
 8
ABSTRACT INTERFACE
 9
 SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
 10
 attribute_val_in, attribute_val_out, flag, ierror)
 11
 TYPE(MPI_Comm) :: oldcomm
 12
 INTEGER :: comm_keyval, ierror
 13
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 14
 attribute_val_out
 15
 LOGICAL :: flag
 16
 17
ABSTRACT INTERFACE
 18
 SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
 19
 attribute_val, extra_state, ierror)
 TYPE(MPI_Comm) :: comm
 20
 21
 INTEGER :: comm_keyval, ierror
 22
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 23
 The copy and delete function arguments to MPI_Win_create_keyval should be declared
 24
according to:
 25
ABSTRACT INTERFACE
 26
 SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
 27
 attribute_val_in, attribute_val_out, flag, ierror)
 28
 TYPE(MPI_Win) :: oldwin
 29
 INTEGER :: win_keyval, ierror
 30
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 31
 attribute_val_out
 32
 LOGICAL :: flag
 33
 34
ABSTRACT INTERFACE
 35
 SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
 36
 extra_state, ierror)
 37
 TYPE(MPI_Win) :: win
 38
 INTEGER :: win_keyval, ierror
 39
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
 40
 The copy and delete function arguments to MPI_Type_create_keyval should be declared
 41
according to:
 42
ABSTRACT INTERFACE
 43
 SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
 44
 attribute_val_in, attribute_val_out, flag, ierror)
 45
 TYPE(MPI_Datatype) :: oldtype
 46
 INTEGER :: type_keyval, ierror
 47
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
 48
```

1attribute_val_out  $\mathbf{2}$ LOGICAL :: flag 3 ABSTRACT INTERFACE 4 SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval, 5attribute_val, extra_state, ierror) 6 TYPE(MPI_Datatype) :: datatype 7 INTEGER :: type_keyval, ierror 8 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state 9 10 The handler-function argument to MPI_Comm_create_errhandler should be declared 11 like this: 12ABSTRACT INTERFACE 13SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) 14TYPE(MPI_Comm) :: comm 15INTEGER :: error_code 16The handler-function argument to MPI_Win_create_errhandler should be declared like 17this: 18 ABSTRACT INTERFACE 19 SUBROUTINE MPI_Win_errhandler_function(win, error_code) 20TYPE(MPI_Win) :: win 21INTEGER :: error_code 22 23The handler-function argument to MPI_File_create_errhandler should be declared like  24 this: 25ABSTRACT INTERFACE 26SUBROUTINE MPI_File_errhandler_function(file, error_code) 27TYPE(MPI_File) :: file 28INTEGER :: error_code 29 The query, free, and cancel function arguments to MPI_Grequest_start should be de-30 clared according to:  31 ABSTRACT INTERFACE 32 SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror) 33 TYPE(MPI_Status) :: status 34 INTEGER :: ierror 35 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 36 37 ABSTRACT INTERFACE 38 SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) 39 INTEGER :: ierror 40INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 41 ABSTRACT INTERFACE 42SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror) 43 INTEGER :: ierror 44 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 45 LOGICAL :: complete 4647 The extent and conversion function arguments to MPI_Register_datarep should be de-48

```
1
clared according to:
 2
ABSTRACT INTERFACE
 3
 SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
 ierror)
 4
 TYPE(MPI_Datatype) :: datatype
 5
 6
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
 INTEGER :: ierror
 7
ABSTRACT INTERFACE
 9
 SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
 10
 filebuf, position, extra_state, ierror)
 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
 18
 19
Fortran Bindings with mpif.h or the mpi Module
 20
With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
 21
subroutines should be declared.
 22
 The user-function argument to MPI_OP_CREATE should be declared like this:
 23
 24
SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE)
 25
 <type> INVEC(LEN), INOUTVEC(LEN)
 26
 INTEGER LEN, DATATYPE
 27
 The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
 28
 29
declared like these:
 30
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
 31
 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
 32
 INTEGER OLDCOMM, COMM_KEYVAL, IERROR
 33
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
 34
 ATTRIBUTE_VAL_OUT
 35
 LOGICAL FLAG
 36
 37
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
 38
 EXTRA_STATE, IERROR)
 39
 INTEGER COMM, COMM_KEYVAL, IERROR
 40
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
 41
 42
 The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be
 43
declared like these:
 44
 45
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
 46
 ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
 47
 INTEGER OLDWIN, WIN_KEYVAL, IERROR
 48
```

$\frac{1}{2}$	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
3	ATTRIBUTE_VAL_OUT LOGICAL FLAG
4	LUGICAL FLAG
5	SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
6	EXTRA_STATE, IERROR)
7	INTEGER WIN, WIN_KEYVAL, IERROR
8	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
9	
10	The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be
11	declared like these:
12	
13	SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
14	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
15	INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
16	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
17	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
18	LOGICAL FLAG
19	SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
20 21	EXTRA_STATE, IERROR)
21	INTEGER DATATYPE, TYPE_KEYVAL, IERROR
23	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
24	
25	The handler-function argument to $MPI_COMM_CREATE_ERRHANDLER$ should be de-
26	clared like this:
27	
28	SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE
29	INTEGER COMM, ERROR_CODE
30	The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-
31	clared like this:
32 33	
34	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
35	INTEGER WIN, ERROR_CODE
36	The handler function argument to MDI FILE CDEATE EDDUANDLED charally be de-
37	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:
38	clared like this.
39	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
40	INTEGER FILE, ERROR_CODE
41	
42	The query, free, and cancel function arguments to MPI_GREQUEST_START should be
43	declared like these:
44	
45	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR
46 47	INTEGER STATUS(MPT_STATUS_STZE), TERROR INTEGER(KIND=MPT_ADDRESS_KIND) EXTRA_STATE
47	TWIPOPUUTUD-ULT_VDUPOPUTUD) EVIUV_DIVIE

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)	1
INTEGER IERROR	2 3
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	4
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)	5
INTEGER IERROR	6
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	7
LOGICAL COMPLETE	8
The extent of a conversion function community to MDL DECISTED DATADED about	9
The extent and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:	10 11
be declated like these.	12
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)	13
INTEGER DATATYPE, IERROR	14
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE	15
	16
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,	17
POSITION, EXTRA_STATE, IERROR)	18
<type> USERBUF(*), FILEBUF(*) INTEGER COUNT, DATATYPE, IERROR</type>	19
INTEGER (KIND=MPI_OFFSET_KIND) POSITION	20 21
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	21
	23
A.1.4 Deprecated Prototype Definitions	24
	25
The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.	26
	27
<pre>/* prototypes for user-defined functions */</pre>	28
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,	29
<pre>void *extra_state, void *attribute_val_in,</pre>	30 31
<pre>void *attribute_val_out, int *flag);</pre>	32
<pre>typedef int MPI_Delete_function(MPI_Comm comm, int keyval,</pre>	33
<pre>void *attribute_val, void *extra_state);</pre>	34
The following are deprecated Fortran user-defined callback subroutine prototypes. The	35
deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-	36
clared like these:	37
	38
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,	39
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)	40 41
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	41
ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG	43
LUGICAL FLAG	44
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)	45
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	46
	47

#### 1A.1.5 Info Keys

- $\mathbf{2}$ The following info keys are reserved. They are strings. 3
- access_style 4
- accumulate_ops  $\mathbf{5}$
- accumulate_ordering 6
- alloc_shared_noncontig  $\overline{7}$
- appnum 8
- arch 9
- cb_block_size 10
- cb_buffer_size 11
- cb_nodes 12
- chunked_item 13
- chunked_size 14
- chunked 15
- collective_buffering 16
- file_perm 17
- filename 18
- file 19
- host 20
- io_node_list 21
- ip_address 22
- ip_port 23
- nb_proc 24
- no_locks 25
- num_io_nodes 26
- path 27
- same_disp_unit 28
- same_size 29
- soft 30
- striping_factor 31
- striping_unit 32wdir
- 33
- 34
- 35
- A.1.6 Info Values 36
- 37The following info values are reserved. They are strings. 38
- false 39
- random 40
- rar  41
- raw 42
- read_mostly 43
- read_once 44
- reverse_sequential 45
- same_op 46
- same_op_no_op 47
- sequential 48

true	1
war	2
waw	3
write_mostly	4
write_once	5
	6
	7
A.2 Summary of the Semantics of all Communicating MPI Routines	8
	9
Stages: i=initialization, s=starting, c=completion, f=freeing	10
Cpl: c=complete, ic=incomplete	11
Loc: I=local, nI=non-local Bold: exceptions, e.g., ic+nI = incomplete+non-local, and c+I = complete+local (both are defined as blocking)	12
<b>Bik:</b> b=blocking, nb=nonblocking. Note that from a user's view point, this column is only a hint. Relevant is, whe-	13
ther a routine is local or not and which resources are blocked until when. See both previous and last columns.	14
Bold: exceptions, e.g., nonblocking precudures without prefix "I", or that "I"only marks immediate return.	15
<b>Op:</b> part of operation type: b-op = blocking operation,	16
nb-op = nonblocking operation, p-op = persistent operation	17
Collective procedures:	18
- C = all processes of the group must call the procedure	19
- sq = in the same sequence	20
- S1 = blocking synchronization, S2 = start-complete-synchronization	20
<b>Blocked resources:</b> They are blocked after the call until the end of subsequent stage where this resource is not further mentioned.	21
futtier mentioneu.	22
Remarks:	23 24
1) Must not return before the corresponding MPI_receive operation is started.	
<ol><li>In a correct MPI program, a call to MPI_(I)RSEND requires that the receiver has</li></ol>	25
already started the corresponding receive. Under this asumption, the call is local.	26
3) Usually, MPI_Wait is non-local, but in this case it is local.	27
4) In case of a MPI_(I)BARRIER, the S1/S2 synchronization is required (instead of "may or may not").	28
5) Collective: all processes must complete, but with the free choice of using MPI_WAIT	29
or MPI_TEST returning flag=TRUE. 6) It also may not return until MPI_INIT was called in the children.	30
7) Addresses are cached on the request handle.	31
8) One of the rare cases that an incomplete call is non-local and therefore blocking.	32
9) One shall not free or deallocate the buffer before the operation is freed, that is MPI_REQUEST_FREE returned.	33
10) For MPI_WAIT and MPI_TEST, see corresponding lines for a) MPI_BSEND, or b) MPI_IBCAST.	34
11) The prefix "I" marks only that this procedure returns immediately. It is not incomplete.	35
<b>12)</b> One of the exceptions that a blocking procedure is local	36
13) It is complete because it is not associated with an operation.	37
14) Nonblocking procedure without an "I" prefix	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

Procedure	Stages	Cpl	Loc	Blk	Ор	Collective			Blocked resources
						C	sq	S1/2	and remarks
MPI_SEND	i-s-c-f	с	nl	b	b-op	-			
MPI_SSEND	i-s-c-f	с	nl	b	b-op	-			1)
MPI_RSEND	i-s-c-f	С	I	b	b-op	-			2) <b>12)</b>
MPI_BSEND	i-s-c-f	С	L	b	b-op	-			12)
MPI_RECV	i-s-c-f	с	nl	b	b-op	-			
MPI_ISEND, MPI_ISSEND	i-s	ic	I	nb	nb-op	-			buffer
MPI_IRECV	i-s	ic	I	nb	nb-op	-			buffer
corresponding MPI_Wait	c-f	с	nl		nb-op	-			
corr. MPI_TEST returning flag=TRUE	c-f	с	1		nb-op	-			
corr. MPI_TEST returning flag=FALSE			I		nb-op	-			buffer cached on req
MPI_IBSEND	i-s	ic	I	nb	nb-op	-			buffer
MPI_IRSEND	i-s	ic	Ι	nb	nb-op	-			buffer 2)
corresponding MPI_Wait	c-f	с	I		nb-op	-			3)
corr. MPI_TEST returning flag=TRUE	c-f	с	1		nb-op	-			
corr. MPI_TEST returning flag=FALSE			I		nb-op	-			buffer 7)
MPI_PROBE	i-s-c-f	с	nl	b	b-op	-			
MPI_IPROBE	i-s-c-f	с	I	b		-			<b>11) 12)</b> 13)
MPI_RECV of a probed message	i-s-c-f	С	I	b	b-op	-			12)
MPI_IRECV of a probed message	i-s	ic	I	nb	nb-op	-			buffer
corresponding MPI_Wait		с	I		nb-op	-			3)
corr. MPI_TEST returning flag=TRUE	c-f	с	Ι		nb-op	-			
corr. MPI_TEST returning flag=FALSE			1		nb-op	-			buffer 7)
MPI_MPROBE	i-s-c-f	ic	nl	b	b-op	-			the message itself <mark>8)</mark>
MPI_IMPROBE	i-s-c-f	ic	I	nb	b-op	-			the message itself
MPI_MRECV of a probed message	i-s-c-f	С	I	b	b-op	-			12)
MPI_IMRECV of a probed message	i-s	ic	I	nb	nb-op	-			buffer
corresponding MPI_Wait	c-f	с	I		nb-op	-			3)
corr. MPI_TEST returning flag=TRUE	c-f	с	I		nb-op	-			
corr. MPI_TEST returning flag=FALSE			I		nb-op	-			buffer 7)
MPI_(- S R)SEND_INIT, MPI_RECV_INIT	i	ic	I	nb	p-op	-			buffer address 14)
corresponding MPI_START, MPI_STARTALL	S	ic	1	nb	p-op	-			buffer address+content 7),1
corresponding MPI_Wait (for RSEND request)	C	ic	I		р-ор	-			buffer address 3), 7)
corresponding MPI_Wait (for other request)	C		nl		р-ор	-			buffer address 7)
corr. MPI_TEST returning flag=TRUE	C		I		p-op	-			buffer address 7)
corr. MPI_TEST returning flag=FALSE			1		p-op	-			buffer content+address 7)
corr. MPI_REQUEST_FREE (for inactive req-handle)	f		I		р-ор	-			
MPI_BSEND_INIT	i	ic	I	nb	p-op	-			buffer address 9), 14)
corresponding MPI_START, MPI_STARTALL	S	ic	I	nb	p-op	-			buffer address+content 7),1
corresponding MPI_Wait	C		I		р-ор	-			buffer address 3,7,9)
corr. MPI_TEST returning flag=TRUE	C		1		p-op	-			buffer address 7,9)
corr. MPI_TEST returning flag=FALSE			1		p-op	-			buffer address+content 7)
corr. MPI_REQUEST_FREE (for inactive req-handle)	f		I		р-ор	-			
MPI_CANCEL of nonblock./persistent pt-to-pt			I		p-op	-			
MPI_SENDRECV(_REPLACE)	i-s-c-f	c	nl	b	b-op	1_			

## A.2. SUMMARY OF THE SEMANTICS OF ALL COMMUNICATING ROUTINES 721

Procedure	Stages	Cpl	Loc	Blk	Ор		1	Blocked resources	
						C sq	S1/2	and remarks	
MPI_BCAST and others	i-s-c-f	с	nl	b	b-op	C sq	S1	4)	
MPI_IBCAST and others	i-s	ic	I	nb	nb-op	C sq		buffer 4)	
MPI_IGATHERV and otherV /W	i-s	ic	1	nb	nb-op	C sq		buffer, array arguments	
corresponding MPI_Wait	c-f	с	nl		nb-op	С	S2	4) 5)	
corr. MPI_TEST returning flag=TRUE	c-f	с	I		nb-op	С	S2	4) 5)	
corr. MPI_TEST returning flag=FALSE			L		nb-op			buffer, array arguments 7)	
MPI_BCAST_INIT and others	i	ic	nl	b	p-op	C sq	S1	buffer address <mark>8)</mark> 9)	
MPI_GATHERV_INIT and otherV/W_INIT	i	ic	nl	b	p-op	C sq	S1	buffer address,	
								array arguments <mark>8)</mark> 9)	
corresponding MPI_START, MPI_STARTALL	s	ic	1	nb	р-ор	C		buffer addr.+content, 4,7),14)	
corresponding MPI_Wait	C		nl		р-ор	С	S2	buffer address and array	
								arguments cached on the	
								reqest handle, 4,5,7,9)	
corr. MPI_TEST returning flag=TRUE	C		1		р-ор	С	S2	buf-addr&arr-args 4,5,7,9)	
corr. MPI_TEST returning flag=FALSE			1		р-ор			buf addr+content&arr-args 7)	
corr. MPI_REQUEST_FREE	f		1		р-ор				
MPI_COMM_CREATE	i-s-c	с	nl	b	b-op	C sq	\$1	coll. over comm arg.	
	i-s-c	c c		b		C sq		coll. over group arg.	
MPI_COMM_DUP, MPI_COMM_DUP_WITH_INFC				_		<u> </u>			
MPI_COMM_DOP, MPI_COMM_DOP_WITH_INFO									
MPI_CART_CREATE, MPI_CRAATI_CREATE MPI_CART_SUB: see MPI_COMM_CREATE							, IVIFI_	DIST_GRAFTI_CREATE,	
MPT_CART_SOB. See MPT_COMM_CREATE	icc	с	nl	b	b-op	C sq	C1	coll. over union of	
MPI INTERCOMM MERGE	i-s-c	C		D	n-oh	C Sq	51	local & remote group	
		:	1	un İn	سم مام	<u> </u>			
MPI_COMM_IDUP	i-s	•••••••		an	nb-op		***	communicator handle	
corresponding MPI_Wait		c	nl		nb-op		S2 S2	5)	
corr. MPI_TEST returning flag=TRUE	c-f	C	1		nb-op	C	52	5)	
corr. MPI_TEST returning flag=FALSE	f		1 10	<b>I</b> a	nb-op	<u> </u>	61		
MPI_COMM_FREE	T	с	nl	b	b-op	C sq	51	see, 6.4.3, Adv. to impl.	
	i-s-c-f	с	nl	b	b-op	C sq	S1	collective over	
MPI_INIT, MPI_INIT_IHREAD								MPI_COMM_WORLD	
	i-s-c-f	с	nl	b	b-op	C sq	S1	collective over all connected	
	i-s-c-f	с	nl	b	b-op	C sq	S1		
MPI_FINALIZE								collective over all connected processes	
MPI_FINALIZE	i-s-c-f i-s-c-f							collective over all connected processes collective over	
MPI_FINALIZE	i-s-c-f	с	nl	b	b-op	C sq	S1	collective over all connected processes collective over comm, 6)	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE		с					S1	collective over all connected processes collective over	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT	i-s-c-f	c c	nl	b b	b-op	C sq	S1	collective over all connected processes collective over comm, 6)	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT MPI_PUT, MPI_GET, MPI_ACCUMULATE	i-s-c-f i-s-c-f	c c	nl nl	b b	b-op b-op	C sq	S1	collective over all connected processes collective over comm, 6) collective over comm	
MPI_FINALIZE  MPI_COMM_SPAWN,MULTIPLE  MPI_COMM_ACCEPT, MPI_COMM_CONNECT  MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures	i-s-c-f i-s-c-f s	c c ic	nl nl l	b b nb	b-op b-op nb-op	C sq	S1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter	
MPI_FINALIZE  MPI_COMM_SPAWN,MULTIPLE  MPI_COMM_ACCEPT, MPI_COMM_CONNECT  MPI_PUT, MPI_GET, MPI_ACCUMULATE  Other one-sided procedures  MPI_FILE_READ/WRITE[_AT SHARED],	i-s-c-f i-s-c-f	c c ic	nl nl	b b	b-op b-op	C sq	S1	collective over all connected processes collective over comm, 6) collective over comm buffer 14)	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures MPI_FILE_READ/WRITE[_AT SHARED], MPI_FILE_DELETE/SEEK/GET_VIEW	i-s-c-f i-s-c-f i-s-c-f	c c ic c	nl nl l	b b nb	b-op b-op nb-op b-op	C sq -	\$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter	
MPI_FINALIZE  MPI_COMM_SPAWN,MULTIPLE  MPI_COMM_ACCEPT, MPI_COMM_CONNECT  MPI_PUT, MPI_GET, MPI_ACCUMULATE  Other one-sided procedures  MPI_FILE_READ/WRITE[_AT]SHARED],  MPI_FILE_READ/WRITE[_AT]_[ALL ORDERED],	i-s-c-f i-s-c-f s	c c ic c	nl nl l	b b nb	b-op b-op nb-op	C sq	\$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter	
MPI_INIT, MPI_INIT_THREAD MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures MPI_FILE_READ/WRITE[_AT]SHARED], MPI_FILE_DELETE/SEEK/GET_VIEW MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_DELETE/SYNC	i-s-c-f i-s-c-f i-s-c-f	c c ic c	nl nl l	b b nb	b-op b-op nb-op b-op	C sq -	\$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter	
MPI_FINALIZE  MPI_COMM_SPAWN,MULTIPLE  MPI_COMM_ACCEPT, MPI_COMM_CONNECT  MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures  MPI_FILE_READ/WRITE[_AT SHARED],  MPI_FILE_DELETE/SEEK/GET_VIEW  MPI_FILE_READ/WRITE[_AT]_[ALL ORDERED],  MPI_FILE_OPEN/CLOSE/SEEK_SHARED,  MPI_FILE_PREALLOCATE/SYNC,	i-s-c-f i-s-c-f i-s-c-f	c c ic c	nl nl l	b b nb	b-op b-op nb-op b-op	C sq -	\$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures MPI_FILE_READ/WRITE[_AT]SHARED], MPI_FILE_DELETE/SEEK/GET_VIEW MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_PREALLOCATE/SYNC, MPI_FILE_SET_VIEW/SIZE/INFO/ATOMICITY	i-s-c-f i-s-c-f i-s-c-f i-s-c-f	c c ic c c	nl nl l nl	b nb b	b-op nb-op b-op b-op	C sq   C sq	\$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter 12)	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures MPI_FILE_READ/WRITE[_AT SHARED], MPI_FILE_DELETE/SEEK/GET_VIEW MPI_FILE_READ/WRITE[_AT]_[ALL ORDERED], MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_PREALLOCATE/SYNC, MPI_FILE_SET_VIEW/SIZE/INFO/ATOMICITY MPI_FILE_IREAD/IWRITE[_AT]SHARED]	i-s-c-f i-s-c-f i-s-c-f i-s-c-f i-s-c-f	c ic c c	nl I I I	b nb b b	b-op nb-op b-op b-op b-op	C sq C sq - - - C sq - -	\$1 \$1 \$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter 12) buffer 10a)	
MPI_FINALIZE MPI_COMM_SPAWN,MULTIPLE MPI_COMM_ACCEPT, MPI_COMM_CONNECT MPI_PUT, MPI_GET, MPI_ACCUMULATE Other one-sided procedures MPI_FILE_READ/WRITE[_AT SHARED], MPI_FILE_DELETE/SEEK/GET_VIEW MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_PREALLOCATE/SYNC, MPI_FILE_SET_VIEW/SIZE/INFO/ATOMICITY	i-s-c-f i-s-c-f i-s-c-f i-s-c-f i-s-c-f i-s	c ic c c	nl nl l	b nb b b	b-op nb-op b-op b-op b-op nb-op	C sq C sq - - - C sq - -	\$1 \$1 \$1	collective over all connected processes collective over comm, 6) collective over comm buffer 14) See corresponding chapter 12)	

1 2	A.3 C Bindings
3	A.3.1 Point-to-Point Communication C Bindings
4 5 6	<pre>int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
7 8	<pre>int MPI_Bsend_init(const void* buf, int count, MPI_Datatype datatype,</pre>
9 10	<pre>int MPI_Buffer_attach(void* buffer, int size)</pre>
11	<pre>int MPI_Buffer_detach(void* buffer_addr, int* size)</pre>
12 13	<pre>int MPI_Cancel(MPI_Request *request)</pre>
14 15	<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>
16 17 18	<pre>int MPI_Ibsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
19 20	int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status)
21 22 23	int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)
24 25	int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
26 27 28	<pre>int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,</pre>
29 30 31	<pre>int MPI_Irsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
32 33	<pre>int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
34 35 36	<pre>int MPI_Issend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>
37 38	int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)
39 40 41	int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)
42	int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
43 44 45	<pre>int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,</pre>
46 47 48	<pre>int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,</pre>

int	MPI_Request_free(MPI_Request *request)	1
int	<pre>MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)</pre>	2 3 4
int	<pre>MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	5 6
int	<pre>MPI_Rsend_init(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)</pre>	7 8 9
int	<pre>MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	10 11
int	<pre>MPI_Send_init(const void* buf, int count, MPI_Datatype datatype,</pre>	12 13 14
int	<pre>MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>	15 16 17 18 19
int	<pre>MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,</pre>	20 21 22
int	<pre>MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>	23 24 25
int	<pre>MPI_Ssend_init(const void* buf, int count, MPI_Datatype datatype,</pre>	26 27
int	<pre>MPI_Startall(int count, MPI_Request array_of_requests[])</pre>	28 29
int	MPI_Start(MPI_Request *request)	30
int	<pre>MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[])</pre>	31 32 33
int	<pre>MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status)</pre>	34 35
int	MPI_Test_cancelled(const MPI_Status *status, int *flag)	36 37
int	MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)	38
int	<pre>MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	39 40 41 42
int	<pre>MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])</pre>	43 44 45
int	<pre>MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)</pre>	45 46 47 48

1	int	MPI_Wait(MPI_Request *request, MPI_Status *status)
2 3 4 5 6	int	<pre>MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>
7 8	A.3.	2 Datatypes C Bindings
9	int	<pre>MPI_Get_address(const void *location, MPI_Aint *address)</pre>
10 11 12	int	<pre>MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>
13 14	int	<pre>MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype, MPI_Count *count)</pre>
15 16 17	int	<pre>MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)</pre>
18 19 20	int	<pre>MPI_Pack_external(const char datarep[], const void *inbuf, int incount,</pre>
21 22 23	int	<pre>MPI_Pack_external_size(const char datarep[], int incount, MPI_Datatype datatype, MPI_Aint *size)</pre>
24 25	int	<pre>MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,</pre>
26 27	int	MPI_Type_commit(MPI_Datatype *datatype)
28 29	int	MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
30 31 32 33 34	int	<pre>MPI_Type_create_darray(int size, int rank, int ndims,</pre>
35 36 37 38	int	<pre>MPI_Type_create_hindexed_block(int count, int blocklength,</pre>
39 40 41	int	<pre>MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],</pre>
42 43 44	int	<pre>MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>
44 45 46 47 48	int	<pre>MPI_Type_create_indexed_block(int count, int blocklength,</pre>

int	MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)	1 $2$
int	<pre>MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>	3 4
	const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_types[], MPI_Datatype *newtype)	5 6
int	<pre>MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>	7 8
	<pre>const int array_of_subsizes[], const int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	9
int	MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)	10 11
int	MPI_Type_free(MPI_Datatype *datatype)	12 13
int	<pre>MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,</pre>	14
	<pre>int max_addresses, int max_datatypes, int array_of_integers[], MPI_Aint array_of_addresses[],</pre>	15 16
	<pre>MPI_Datatype array_of_datatypes[])</pre>	17
int	<pre>MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>	18 19
int	<pre>MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *1b,</pre>	20 21
	MPI_Aint *extent)	22
int	<pre>MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb, MPI_Count *extent)</pre>	23 24 25
int	<pre>MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)</pre>	26 27
int	<pre>MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb, MPI_Count *true_extent)</pre>	28 29 30
int	<pre>MPI_Type_indexed(int count, const int array_of_blocklengths[],</pre>	31 32 33
int	MPI_Type_size(MPI_Datatype datatype, int *size)	34 35
int	MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)	36
int	<pre>MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	37 38 39
int	<pre>MPI_Unpack(const void* inbuf, int insize, int *position, void *outbuf,</pre>	40 41
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>	42 43 44 45
MPI.	_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)	46
MPI.	_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)	47 48

1A.3.3 Collective Communication C Bindings  $\mathbf{2}$ int MPI_Allgather(const void* sendbuf, int sendcount, 3 MPI_Datatype sendtype, void* recvbuf, int recvcount, 4 MPI_Datatype recvtype, MPI_Comm comm) 56 int MPI_Allgather_init(const void* sendbuf, int sendcount, 7 MPI_Datatype sendtype, void* recvbuf, int recvcount, 8 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 9 MPI_Request *request) 10 int MPI_Allgatherv(const void* sendbuf, int sendcount, 11 MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 12const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 13 14int MPI_Allgatherv_init(const void* sendbuf, int sendcount, 15MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 16 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 17MPI_Info info, MPI_Request* request) 18 int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, 19 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 2021int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, 22MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 23MPI_Info info, MPI_Request *request) 24int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 25void* recvbuf, int recvcount, MPI_Datatype recvtype, 26MPI_Comm comm) 2728int MPI_Alltoall_init(const void* sendbuf, int sendcount, 29 MPI_Datatype sendtype, void* recvbuf, int recvcount, 30 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 31MPI_Request *request) 32 int MPI_Alltoallv(const void* sendbuf, const int sendcounts[], 33 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 34 const int recvcounts[], const int rdispls[], 35 MPI_Datatype recvtype, MPI_Comm comm) 36 37 int MPI_Alltoallv_init(const void* sendbuf, const int sendcounts[], 38 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 39 const int recvcounts[], const int rdispls[], 40 MPI_Datatype recvtype, MPI_Comm comm, MPI_info info, 41 MPI_Request *request) 42int MPI_Alltoallw(const void* sendbuf, const int sendcounts[], 43 const int sdispls[], const MPI_Datatype sendtypes[], 44 void* recvbuf, const int recvcounts[], const int rdispls[], 45 const MPI_Datatype recvtypes[], MPI_Comm comm) 46 47int MPI_Alltoallw_init(const void* sendbuf, const int sendcounts[], 48

	<pre>const int sdispls[], const MPI_Datatype sendtypes[],</pre>	1
	<pre>void* recvbuf, const int recvcounts[], const int rdispls[],</pre>	2
	<pre>const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,</pre>	3
	MPI_Request *request)	4
int MPI_Barr	ier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)	5 6
	ier(MPI_Comm comm)	7
IIIC MII_DAII		8
int MPI_Bcas	t_init(void* buffer, int count, MPI_Datatype datatype,	9
	<pre>int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	10
int MPI_Bcas	t(void* buffer, int count, MPI_Datatype datatype, int root,	11
	MPI_Comm comm)	12
int MDT Errag	an (const worldw condbuf worldw no whuf int count	13
IIIC MFI_EXSC	an(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	14
	Mri_Datatype datatype, Mri_op op, Mri_Comm comm)	15
int MPI_Exsc	an_init(const void* sendbuf, void* recvbuf, int count,	16
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	17 18
	MPI_Info info, MPI_Request *request)	18
int MPI Gath	er(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	20
	void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,	21
	MPI_Comm comm)	22
int MPT Cath	er_init(const void* sendbuf, int sendcount,	23
Int mi_dath	MPI_Datatype sendtype, void* recvbuf, int recvcount,	24
	MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	25
	MPI_Request *request)	26
		27
int MPI_Gath	erv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	28
	<pre>void* recvbuf, const int recvcounts[], const int displs[],</pre>	29
	MPI_Datatype recvtype, int root, MPI_Comm comm)	30
int MPI_Gath	erv_init(const void* sendbuf, int sendcount,	31
	<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],</pre>	32
	<pre>const int displs[], MPI_Datatype recvtype, int root,</pre>	33
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	34
int MDT Tall	gather(const void* sendbuf, int sendcount,	35 36
IIIC MEI_IAII	MPI_Datatype sendtype, void* recvbuf, int recvcount,	30 37
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	38
		39
int MPI_Iall	gatherv(const void* sendbuf, int sendcount,	40
	<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],</pre>	41
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	42
	MPI_Request* request)	43
int MPI_Iall	reduce(const void* sendbuf, void* recvbuf, int count,	44
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	45
	MPI_Request *request)	46
int MDT T		47
IIIC MFI_IALL	<pre>toall(const void* sendbuf, int sendcount,</pre>	48

1MPI_Datatype sendtype, void* recvbuf, int recvcount,  $\mathbf{2}$ MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 3 int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], 4 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 5const int recvcounts[], const int rdispls[], 6 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 7 8 int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], 9 const int sdispls[], const MPI_Datatype sendtypes[], 10void* recvbuf, const int recvcounts[], const int rdispls[], 11 const MPI_Datatype recvtypes[], MPI_Comm comm, 12MPI_Request *request) 13 int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) 1415int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root, 16MPI_Comm comm, MPI_Request *request) 17int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count, 18 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 19 MPI_Request *request) 2021int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 22 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, 23MPI_Comm comm, MPI_Request *request) 24int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 25void* recvbuf, const int recvcounts[], const int displs[], 26MPI_Datatype recvtype, int root, MPI_Comm comm, 27MPI_Request *request) 2829 int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count, 30 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, 31MPI_Request *request) 32 int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf, 33 int recvcount, MPI_Datatype datatype, MPI_Op op, 34 MPI_Comm comm, MPI_Request *request) 35 36 int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, 37 const int recvcounts[], MPI_Datatype datatype, MPI_Op op, 38 MPI_Comm comm, MPI_Request *request) 39 int MPI_Iscan(const void* sendbuf, void* recvbuf, int count, 40MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 41 42MPI_Request *request) 43 int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 44 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, 45MPI_Comm comm, MPI_Request *request) 4647int MPI_Iscatterv(const void* sendbuf, const int sendcounts[], 48 const int displs[], MPI_Datatype sendtype, void* recvbuf,

ANNEX A. LANGUAGE BINDINGS SUMMARY

	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)	$\frac{1}{2}$
int	MPI_Op_commutative(MPI_Op op, int *commute)	$\frac{3}{4}$
int	MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)	5
	MPI_Op_free(MPI_Op *op)	6
		7 8
IIIC	<pre>MPI_Reduce(const void* sendbuf, void* recvbuf, int count,</pre>	9
int	MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,	10
1110	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	11 12 13
int	MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,	14
	MPI_Datatype datatype, MPI_Op op)	15 16
int	MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,	17
	<pre>int recvcount, MPI_Datatype datatype, MPI_Op op,</pre>	18
	MPI_Comm comm)	19 20
int	MPI_Reduce_scatter_block_init(const void* sendbuf, void* recvbuf,	20
	int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)	22
int	MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,	23 24
1110	const int recvcounts[], MPI_Datatype datatype, MPI_Op op,	24
	MPI_Comm comm)	26
int	<pre>MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf,</pre>	27 28
	<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MDI_Comm_commMDI_Info_infoMDI_Dermost tracement)</pre>	29
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	30
int	<pre>MPI_Scan(const void* sendbuf, void* recvbuf, int count,</pre>	31 32
		33
int	<pre>MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,</pre>	34
	MPI_Info info, MPI_Request *request)	$35 \\ 36$
int	MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	37
	void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,	38
	MPI_Comm comm)	39 40
int	<pre>MPI_Scatter_init(const void* sendbuf, int sendcount,</pre>	40
	MPI_Datatype sendtype, void* recvbuf, int recvcount,	42
	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	43
int	MPI_Scatterv(const void* sendbuf, const int sendcounts[],	44 45
1110	const int displs[], MPI_Datatype sendtype, void* recvbuf,	46
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	47
		48

	730	ANNEX A. LANGUAGE BINDINGS SUMMARY
1 2 3 4 5	int MPI_Sca	atterv_init(const void* sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
6 7	A.3.4 Grou	ps, Contexts, Communicators, and Caching C Bindings
8 9	int MPI_Com	<pre>mm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)</pre>
9 10 11	int MPI_Com	m_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm)
12 13 14 15	int MPI_Com	<pre>mm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state)</pre>
16	int MPI_Com	nm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
17 18	int MPI_Com	m_delete_attr(MPI_Comm comm, int comm_keyval)
19 20	int MPI_COM	<pre>M_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
21 22	int MPI_Com	nm_dup(MPI_Comm comm, MPI_Comm *newcomm)
23	int MPI_Com	m_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
24 25	int MPI_Com	m_free_keyval(int *comm_keyval)
26	int MPI_Com	m_free(MPI_Comm *comm)
27 28 29	int MPI_Com	<pre>m_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val, int *flag)</pre>
30	int MPI_Com	<pre>um_get_info(MPI_Comm comm, MPI_Info *info_used)</pre>
31 32	int MPI_Com	<pre>m_get_name(MPI_Comm comm, char *comm_name, int *resultlen)</pre>
33	int MPI_Com	m_group(MPI_Comm comm, MPI_Group *group)
$\frac{34}{35}$	int MPI_Com	m_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
36 37	int MPI_Com	m_idup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm, MPI_Request *request)
38 39 40 41	int MPI_COM	<pre>M_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
42 43 44	int MPI_COM	<pre>M_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void *attribute_val, void *extra_state)</pre>
45	int MPI_Com	m_rank(MPI_Comm comm, int *rank)
$46 \\ 47$	int MPI_Com	m_remote_group(MPI_Comm comm, MPI_Group *group)
48	int MPI_Com	m_remote_size(MPI_Comm comm, int *size)

<pre>int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribu</pre>	
<pre>int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)</pre>	2 3
<pre>int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)</pre>	4
	5
<pre>int MPI_Comm_size(MPI_Comm comm, int *size)</pre>	6
<pre>int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *new</pre>	comm) 7 8
<pre>int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,</pre>	9
MPI_Info info, MPI_Comm *newcomm)	10
<pre>int MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>	11 12
<pre>int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *resul</pre>	
<pre>int MPI_Group_difference(MPI_Group group1, MPI_Group group2,</pre>	14
MPI_Group *newgroup)	15 16
<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[],</pre>	17
MPI_Group *newgroup)	18
<pre>int MPI_Group_free(MPI_Group *group)</pre>	19 20
<pre>int MPI_Group_incl(MPI_Group group, int n, const int ranks[],</pre>	20
MPI_Group *newgroup)	22
<pre>int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,</pre>	23
MPI_Group *newgroup)	24 25
<pre>int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],</pre>	26
MPI_Group *newgroup)	27
<pre>int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],</pre>	28 29
MPI_Group *newgroup)	30
<pre>int MPI_Group_rank(MPI_Group group, int *rank)</pre>	31
int MPI_Group_size(MPI_Group group, int *size)	32 33
int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ra	
MPI_Group group2, int ranks2[])	35
<pre>int MPI_Group_union(MPI_Group group1, MPI_Group group2,</pre>	36 37
MPI_Group *newgroup)	38
<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	39
MPI_Comm peer_comm, int remote_leader, int tag,	40 41
MPI_Comm *newintercomm)	41 42
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,	43
MPI_Comm *newintracomm)	44
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_a	ttr_fn, 45 46
<pre>MPI_Type_delete_attr_function *type_delete_attr_fn, int *type_keyval, void *extra_state)</pre>	47
	48

1 int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)  $\mathbf{2}$ int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, 3 void *extra_state, void *attribute_val_in, 4 void *attribute_val_out, int *flag) 56 int MPI_Type_free_keyval(int *type_keyval) 7 int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, 8 void *attribute_val, int *flag) 9 10int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, 11 int *resultlen) 12int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval, 13 void *extra_state, void *attribute_val_in, 14void *attribute_val_out, int *flag) 1516int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, 17void *attribute_val, void *extra_state) 18 int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval, 19 void *attribute_val) 2021int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name) 22int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, 23MPI_Win_delete_attr_function *win_delete_attr_fn,  24 int *win_keyval, void *extra_state) 2526int MPI_Win_delete_attr(MPI_Win win, int win_keyval) 27int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state, 28void *attribute_val_in, void *attribute_val_out, int *flag) 2930 int MPI_Win_free_keyval(int *win_keyval) 31int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val, 3233 int *flag) 34 int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen) 35 36 int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, 37 void *attribute_val_in, void *attribute_val_out, int *flag) 38 int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, 39 void *attribute_val, void *extra_state) 4041int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val) 42int MPI_Win_set_name(MPI_Win win, const char *win_name) 43 44 45A.3.5 Process Topologies C Bindings 46 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]) 4748

int	<pre>MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],</pre>	1 2
int	MPI_Cartdim_get(MPI_Comm comm, int *ndims)	$\frac{3}{4}$
int	<pre>MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[])</pre>	5 6
int	<pre>MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>	7 8 9
int	MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)	10
int	<pre>MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>	11 12 13
int	MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)	14
int	MPI_Dims_create(int nnodes, int ndims, int dims[])	15 16
int	<pre>MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>	17 18 19 20 21
int	<pre>MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>	22 23 24 25
int	<pre>MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>	26 27 28
int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	29 30 31
int	<pre>MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	32 33 34
int	MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)	35
int	<pre>MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[])</pre>	36 37 38
int	<pre>MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],</pre>	39 40
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	$41 \\ 42$
int	<pre>MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	43 44
int	<pre>MPI_Ineighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	45 46 47 48

1int MPI_Ineighbor_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 MPI_Request *request) 5int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount, 6 MPI_Datatype sendtype, void* recvbuf, int recvcount, 7 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 8 9 int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[], 10const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 11 const int recvcounts[], const int rdispls[], 12MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 13 int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[], 14 const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], 15void* recvbuf, const int recvcounts[], 16const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], 17 MPI_Comm comm, MPI_Request *request) 18 19 int MPI_Neighbor_allgather(const void* sendbuf, int sendcount, 20MPI_Datatype sendtype, void* recvbuf, int recvcount, 21MPI_Datatype recvtype, MPI_Comm comm) 22 int MPI_Neighbor_allgather_init(const void* sendbuf, int sendcount, 23MPI_Datatype sendtype, void* recvbuf, int recvcount, 24MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 25MPI_Request *request) 2627int MPI_Neighbor_allgatherv(const void* sendbuf, int sendcount, 28MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 29const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 30 int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount,  31 MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 32 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 33 MPI_Info info, MPI_Request *request) 34 35 int MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, 36 MPI_Datatype sendtype, void* recvbuf, int recvcount, 37 MPI_Datatype recvtype, MPI_Comm comm) 38 int MPI_Neighbor_alltoall_init(const void* sendbuf, int sendcount, 39 MPI_Datatype sendtype, void* recvbuf, int recvcount, 40MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 41 MPI_Request *request) 4243 int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[], 44 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 45const int recvcounts[], const int rdispls[], 46MPI_Datatype recvtype, MPI_Comm comm) 47int MPI_Neighbor_alltoallv_init(const void* sendbuf, 48

ANNEX A. LANGUAGE BINDINGS SUMMARY

<pre>const int sendcounts[], const int sdispls[],</pre>	1
<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],</pre>	2
<pre>const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MDI_Lafa_infaMDI_Datatype recvtype, MPI_Comm comm,</pre>	3 4
MPI_Info info, MPI_Request *request)	5
<pre>int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>	6
<pre>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],</pre>	7
<pre>void* recvbuf, const int recvcounts[], const MDL Aist mdignla[] const MDL Detetune recutumes[]</pre>	8
const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm)	9
	10 11
int MPI_Neighbor_alltoallw_init(const void* sendbuf,	11
<pre>const int sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void* recvbuf,</pre>	13
const int recvcounts[], const MPI_Aint rdispls[],	14
<pre>const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,</pre>	15
MPI_Request *request)	16
int MPI_Topo_test(MPI_Comm comm, int *status)	17
	18 19
	20
A.3.6 MPI Environmental Management C Bindings	21
<pre>int MPI_Abort(MPI_Comm comm, int errorcode)</pre>	22
<pre>int MPI_Add_error_class(int *errorclass)</pre>	23 24
<pre>int MPI_Add_error_code(int errorclass, int *errorcode)</pre>	25
<pre>int MPI_Add_error_string(int errorcode, const char *string)</pre>	26 27
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)	28
<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)</pre>	29 30
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function	31
<pre>*comm_errhandler_fn, MPI_Errhandler *errhandler)</pre>	32
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)	33
	34 35
<pre>int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)</pre>	36
<pre>int MPI_Errhandler_free(MPI_Errhandler *errhandler)</pre>	37
<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>	38 39
<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>	40
<pre>int MPI_File_call_errhandler(MPI_File fh, int errorcode)</pre>	41 42
int MPI_File_create_errhandler(MPI_File_errhandler_function	42
*file_errhandler_fn, MPI_Errhandler *errhandler)	44
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)	45
-	46
<pre>int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)</pre>	47
	48

1 int MPI_Finalized(int *flag)  $\mathbf{2}$ int MPI_Finalize(void) 3 4 int MPI_Free_mem(void *base) 5int MPI_Get_library_version(char *version, int *resultlen) 6  $\overline{7}$ int MPI_Get_processor_name(char *name, int *resultlen) 8 int MPI_Get_version(int *version, int *subversion) 9 10int MPI_Initialized(int *flag) 11 int MPI_Init(int *argc, char ***argv) 1213int MPI_Win_call_errhandler(MPI_Win win, int errorcode) 14int MPI_Win_create_errhandler(MPI_Win_errhandler_function 15*win_errhandler_fn, MPI_Errhandler *errhandler) 1617int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler) 18 int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler) 1920double MPI_Wtick(void) 21double MPI_Wtime(void) 2223 24 A.3.7 The Info Object C Bindings 2526int MPI_Info_create(MPI_Info *info) 27int MPI_Info_delete(MPI_Info info, const char *key) 2829int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo) 30 int MPI_Info_free(MPI_Info *info)  31 32 int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value, 33 int *flag) 34int MPI_Info_get_nkeys(MPI_Info info, int *nkeys) 3536 int MPI_Info_get_nthkey(MPI_Info info, int n, char *key) 37 int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen, 38int *flag) 39 40int MPI_Info_set(MPI_Info info, const char *key, const char *value) 41 4243A.3.8 Process Creation and Management C Bindings 44int MPI_Close_port(const char *port_name) 4546int MPI_Comm_accept(const char *port_name, MPI_Info info, int root, 47MPI_Comm comm, MPI_Comm *newcomm) 48

int MPI_Comm	_connect(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)	1 $2$
int MPI_Comm	_disconnect(MPI_Comm *comm)	3
		4 5
		6
int MPI_Comm	_join(int fd, MPI_Comm *intercomm)	7
int MPI_Comm	_spawn(const char *command, char *argv[], int maxprocs,	8
	MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,	9 10
	<pre>int array_of_errcodes[])</pre>	11
int MPI_Comm	_spawn_multiple(int count, char *array_of_commands[],	12
	<pre>char **array_of_argv[], const int array_of_maxprocs[],</pre>	13
	const MPI_Info array_of_info[], int root, MPI_Comm comm,	14
	<pre>MPI_Comm *intercomm, int array_of_errcodes[])</pre>	15
int MPI_Look	up_name(const char *service_name, MPI_Info info,	16 17
	char *port_name)	18
int MPI_Open	_port(MPI_Info info, char *port_name)	19
int MPI_Publ	<pre>ish_name(const char *service_name, MPI_Info info,</pre>	20
	<pre>const char *port_name)</pre>	21
int MPI Unpu	blish_name(const char *service_name, MPI_Info info,	22 23
	const char *port_name)	24
		25
A30 One S	ided Communications C Bindings	26
		27
int MPI_Accu	<pre>mulate(const void *origin_addr, int origin_count,</pre>	28
	MPI_Datatype origin_datatype, int target_rank,	29 30
	MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	31
		32
int MPI_Comp	are_and_swap(const void *origin_addr, const void *compare_addr,	33
	void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)	34
		35
int MPI_Fetc	h_and_op(const void *origin_addr, void *result_addr,	36 37
	<pre>MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win)</pre>	38
	•••	39
int MPI_Get_	accumulate(const void *origin_addr, int origin_count,	40
	MPI_Datatype origin_datatype, void *result_addr,	41
	<pre>int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count,</pre>	42
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	43 $44$
int MDT det (		44 45
INT MPI_Get(	<pre>void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank,</pre>	46
	MPI_Aint target_disp, int target_count,	47
		48

1		MPI_Datatype target_datatype, MPI_Win win)
2	int	MPI_Put(const void *origin_addr, int origin_count,
3 4		MPI_Datatype origin_datatype, int target_rank,
4 5		MPI_Aint target_disp, int target_count,
6		MPI_Datatype target_datatype, MPI_Win win)
7	in+	MDI Decountilate (const word torigin oddr int origin count
8	THC	<pre>MPI_Raccumulate(const void *origin_addr, int origin_count,</pre>
9		MPI_Aint target_disp, int target_count,
10		MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
11		MPI_Request *request)
12		
13	int	MPI_Rget_accumulate(const void *origin_addr, int origin_count,
14		<pre>MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype,</pre>
15		int target_rank, MPI_Aint target_disp, int target_count,
16		MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
17 18		MPI_Request *request)
19		
20	int	MPI_Rget(void *origin_addr, int origin_count,
21		<pre>MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,</pre>
22		MPI_AINT target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win,
23		MPI_Request *request)
24		
25	int	MPI_Rput(const void *origin_addr, int origin_count,
26		MPI_Datatype origin_datatype, int target_rank,
27		MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win,
28		MPI_Request *request)
29 30		
31	int	MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
32		<pre>MPI_Comm comm, void *baseptr, MPI_Win *win)</pre>
33	int	MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
34		MPI_Comm comm, void *baseptr, MPI_Win *win)
35	int	MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
36		
37	int	MPI_Win_complete(MPI_Win win)
38 39	int	MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
40	int	MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
41		MPI_Comm comm, MPI_Win *win)
42	in+	MDT Uin datach (MDT Uin win const word thace)
43	TUC	MPI_Win_detach(MPI_Win win, const void *base)
44	int	MPI_Win_fence(int assert, MPI_Win win)
45 46	int	MPI_Win_flush_all(MPI_Win win)
47	int	MPI_Win_flush(int rank, MPI_Win win)
48		

<pre>int MPI_Win_flush_local_all(MPI_Win win)</pre>	1
int MPI_Win_flush_local(int rank, MPI_Win win)	2 3
int MPI_Win_free(MPI_Win *win)	4
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)	5
<pre>int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)</pre>	6 7
<b>v</b>	8
<pre>int MPI_Win_lock_all(int assert, MPI_Win win)</pre>	9
<pre>int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)</pre>	10 11
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)	12
int MPI_Win_set_info(MPI_Win win, MPI_Info info)	13
int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,	14 15
<pre>int *disp_unit, void *baseptr)</pre>	16
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)	17
int MPI_Win_sync(MPI_Win win)	18 19
int MPI_Win_test(MPI_Win win, int *flag)	20
ũ	21 22
<pre>int MPI_Win_unlock_all(MPI_Win win)</pre>	22
<pre>int MPI_Win_unlock(int rank, MPI_Win win)</pre>	24
int MPI_Win_wait(MPI_Win win)	25 26
	20
A.3.10 External Interfaces C Bindings	28
<pre>int MPI_Grequest_complete(MPI_Request request)</pre>	29 30
<pre>int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,</pre>	31
MPI_Grequest_free_function *free_fn,	32
<pre>MPI_Grequest_cancel_function *cancel_fn, void *extra_state, MPI_Request *request)</pre>	33 34
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)	35
	36
<pre>int MPI_Is_thread_main(int *flag)</pre>	37 38
<pre>int MPI_Query_thread(int *provided)</pre>	39
<pre>int MPI_Status_set_cancelled(MPI_Status *status, int flag)</pre>	40
<pre>int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,</pre>	41 42
	43 44
<pre>int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype, MPI_Count count)</pre>	44 45
	46
	47 48
	48

1	A.3.11 I/O C Bindings
2 3 4	int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state)
5	<pre>int MPI_File_close(MPI_File *fh)</pre>
6 7	<pre>int MPI_File_delete(const char *filename, MPI_Info info)</pre>
8	int MPI_File_get_amode(MPI_File fh, int *amode)
9 10	<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>
11 12 13	<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>
14	<pre>int MPI_File_get_group(MPI_File fh, MPI_Group *group)</pre>
15 16	<pre>int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)</pre>
17	<pre>int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)</pre>
18 19	<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>
20	<pre>int MPI_File_get_size(MPI_File fh, MPI_Offset *size)</pre>
21 22 23	<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,</pre>
24 25	int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)
26 27 28	int MPI_File_iread_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
29 30	<pre>int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>
31 32 33	<pre>int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
34 35	int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
36 37 38	int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
39 40	<pre>int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
41 42 43	<pre>int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
44 45	<pre>int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
46 47 48	int MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	1 2
int	<pre>MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)</pre>	3 4 5
int	MPI_File_preallocate(MPI_File fh, MPI_Offset size)	6
int	<pre>MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	7 8 9
int	MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	10
int	MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	11 12 13
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	14 15 16
int	MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)	17
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	18 19 20
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	21 22
int	<pre>MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	23 24 25
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	26 27
int	MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)	28 29
int	MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	30 31
int	MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	32 33 34
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	35
int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)	$\frac{36}{37}$
int	MPI_File_set_atomicity(MPI_File fh, int flag)	38
int	MPI_File_set_info(MPI_File fh, MPI_Info info)	$\frac{39}{40}$
int	MPI_File_set_size(MPI_File fh, MPI_Offset size)	41
int	MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)	42 43 44
int	MPI_File_sync(MPI_File fh)	45
int	<pre>MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>	46 47 48

```
1
 int MPI_File_write_all_end(MPI_File fh, const void *buf,
\mathbf{2}
 MPI_Status *status)
3
 int MPI_File_write_all(MPI_File fh, const void *buf, int count,
4
 MPI_Datatype datatype, MPI_Status *status)
5
6
 int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
7
 const void *buf, int count, MPI_Datatype datatype)
8
 int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
9
 MPI_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
 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
14
 int count, MPI_Datatype datatype, MPI_Status *status)
15
16
 int MPI_File_write(MPI_File fh, const void *buf, int count,
17
 MPI_Datatype datatype, MPI_Status *status)
18
 int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
19
 MPI_Datatype datatype)
20
21
 int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
22
 MPI_Status *status)
23
 int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
^{24}
 MPI_Datatype datatype, MPI_Status *status)
25
26
 int MPI_File_write_shared(MPI_File fh, const void *buf, int count,
27
 MPI_Datatype datatype, MPI_Status *status)
28
 int MPI_Register_datarep(const char *datarep,
29
 MPI_Datarep_conversion_function *read_conversion_fn,
30
 MPI_Datarep_conversion_function *write_conversion_fn,
31
 MPI_Datarep_extent_function *dtype_file_extent_fn,
32
 void *extra_state)
33
34
35
 A.3.12 Language Bindings C Bindings
36
37
 int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
38
 int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
39
40
 int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
41
 int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
42
43
 int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
44
 int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
45
46
 MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
47
 MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
48
```

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	1
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	2 3
MPI_Fint MPI_File_c2f(MPI_File file)	4
MPI_File MPI_File_f2c(MPI_Fint file)	5
MPI_Fint MPI_Group_c2f(MPI_Group group)	6 7
	8
MPI_Group MPI_Group_f2c(MPI_Fint group)	9 10
MPI_Fint MPI_Info_c2f(MPI_Info info)	11
MPI_Info MPI_Info_f2c(MPI_Fint info)	12
MPI_Fint MPI_Message_c2f(MPI_Message message)	13 14
MPI_Message MPI_Message_f2c(MPI_Fint message)	15
MPI_Fint MPI_Op_c2f(MPI_Op op)	16 17
MPI_Op MPI_Op_f2c(MPI_Fint op)	18
MPI_Fint MPI_Request_c2f(MPI_Request request)	19
MPI_Request MPI_Request_f2c(MPI_Fint request)	20 21
	22
<pre>int MPI_Status_c2f08(const MPI_Status *c_status,</pre>	23
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)	24 25
<pre>int MPI_Status_f082c(const MPI_F08_status *f08_status,</pre>	26
MPI_Status *c_status)	27 28
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)	29
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)	30
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)	31 32
	33
MPI_Fint MPI_Win_c2f(MPI_Win win)	34 25
MPI_Win MPI_Win_f2c(MPI_Fint win)	35 36
	37
A.3.13 Tools / Profiling Interface C Bindings	38 39
<pre>int MPI_Pcontrol(const int level,)</pre>	40
	41
A.3.14 Tools / MPI Tool Information Interface C Bindings	42 43
<pre>int MPI_T_category_changed(int *stamp)</pre>	44
<pre>int MPI_T_category_get_categories(int cat_index, int len, int indices[])</pre>	45 46
<pre>int MPI_T_category_get_cvars(int cat_index, int len, int indices[])</pre>	40
	48

1	int	<pre>MPI_T_category_get_index(const char *name, int *cat_index)</pre>
2 3 4 5	int	<pre>MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>
6 7	int	MPI_T_category_get_num(int *num_cat)
8	int	<pre>MPI_T_category_get_pvars(int cat_index, int len, int indices[])</pre>
9 10	int	<pre>MPI_T_cvar_get_index(const char *name, int *cvar_index)</pre>
11 12 13	int	<pre>MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,</pre>
14 15	int	MPI_T_cvar_get_num(int *num_cvar)
16 17	int	<pre>MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle, MPI_T_cvar_handle *handle, int *count)</pre>
18 19	int	MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
20	int	<pre>MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf)</pre>
21 22	int	MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf)
23 24	int	<pre>MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,</pre>
25 26 27	int	<pre>MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,</pre>
28	int	MPI_T_finalize(void)
29 30	int	MPI_T_init_thread(int required, int *provided)
31	int	<pre>MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)</pre>
32 33 34 35 36	int	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>
37	int	MPI_T_pvar_get_num(int *num_pvar)
38 39 40	int	<pre>MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,</pre>
41 42	int	<pre>MPI_T_pvar_handle_free(MPI_T_pvar_session session, MPI_T_pvar_handle *handle)</pre>
43 44 45	int	<pre>MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle, void* buf)</pre>
46 47 48	int	<pre>MPI_T_pvar_readreset(MPI_T_pvar_session session,</pre>

int	<pre>MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>	1
int	<pre>MPI_T_pvar_session_create(MPI_T_pvar_session *session)</pre>	2 3
int	MPI_T_pvar_session_free(MPI_T_pvar_session *session)	4
int	MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)	5 6
int	MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)	7
int	<pre>MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>	8 9 10
A.3.	15 Deprecated C Bindings	11 12
int	MPI_Attr_delete(MPI_Comm comm, int keyval)	13 14
int	MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)	15
	MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)	16 17
	MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,	18
THE	void *attribute_val_in, void *attribute_val_out, int *flag)	19 20
int	<pre>MPI_Keyval_create(MPI_Copy_function *copy_fn,</pre>	21
	MPI_Delete_function *delete_fn, int *keyval,	22
	void* extra_state)	23 24
int	MPI_Keyval_free(int *keyval)	24
int	MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,	26
	void *attribute_val_in, void *attribute_val_out, int *flag)	27
int	MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,	28 29
	void *extra_state)	30
		31
		32
		33
		34 35
		36
		37
		38
		39
		40
		41 42
		43
		44
		45
		46
		47
		48

```
A.4 Fortran 2008 Bindings with the mpi_f08 Module
1
\mathbf{2}
 A.4.1 Point-to-Point Communication Fortran 2008 Bindings
3
4
 MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
5
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
6
 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
11
 MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
12
 INTEGER, INTENT(IN) :: count, dest, tag
13
14
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 TYPE(MPI_Comm), INTENT(IN) :: comm
15
 TYPE(MPI_Request), INTENT(OUT) :: request
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
 MPI_Buffer_attach(buffer, size, ierror)
19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
20
 INTEGER, INTENT(IN) :: size
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_Buffer_detach(buffer_addr, size, ierror)
24
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
 TYPE(C_PTR), INTENT(OUT) :: buffer_addr
26
 INTEGER, INTENT(OUT) :: size
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
 MPI_Cancel(request, ierror)
29
 TYPE(MPI_Request), INTENT(IN) :: request
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 MPI_Get_count(status, datatype, count, ierror)
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
 MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
38
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
39
 INTEGER, INTENT(IN) :: count, dest, tag
40
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
 TYPE(MPI_Comm), INTENT(IN) :: comm
42
 TYPE(MPI_Request), INTENT(OUT) :: request
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
 MPI_Improbe(source, tag, comm, flag, message, status, ierror)
46
 INTEGER, INTENT(IN) :: source, tag
47
 TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
 LOGICAL, INTENT(OUT) :: flag
 2
 TYPE(MPI_Message), INTENT(OUT) :: message
 3
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 4
 5
MPI_Imrecv(buf, count, datatype, message, request, ierror)
 6
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 7
 INTEGER, INTENT(IN) :: count
 8
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 9
 TYPE(MPI_Message), INTENT(INOUT) :: message
 10
 TYPE(MPI_Request), INTENT(OUT) :: request
 11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 12
 13
MPI_Iprobe(source, tag, comm, flag, status, ierror)
 14
 INTEGER, INTENT(IN) :: source, tag
 15
 TYPE(MPI_Comm), INTENT(IN) :: comm
 16
 LOGICAL, INTENT(OUT) :: flag
 17
 TYPE(MPI_Status) :: status
 18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 19
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
 20
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
 21
 INTEGER, INTENT(IN) :: count, source, tag
 22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 23
 TYPE(MPI_Comm), INTENT(IN) :: comm
 24
 TYPE(MPI_Request), INTENT(OUT) :: request
 25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
 27
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
 28
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
 29
 INTEGER, INTENT(IN) :: count, dest, tag
 30
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 31
 TYPE(MPI_Comm), INTENT(IN) :: comm
 32
 TYPE(MPI_Request), INTENT(OUT) :: request
 33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 34
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
 35
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
 36
 INTEGER, INTENT(IN) :: count, dest, tag
 37
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 38
 TYPE(MPI_Comm), INTENT(IN) :: comm
 39
 TYPE(MPI_Request), INTENT(OUT) :: request
 40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 41
 42
MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
 43
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
 44
 INTEGER, INTENT(IN) :: count, dest, tag
 45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 46
 TYPE(MPI_Comm), INTENT(IN) :: comm
 47
 TYPE(MPI_Request), INTENT(OUT) :: request
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Mprobe(source, tag, comm, message, status, ierror)
3
 INTEGER, INTENT(IN) :: source, tag
4
 TYPE(MPI_Comm), INTENT(IN) :: comm
5
 TYPE(MPI_Message), INTENT(OUT) :: message
6
 TYPE(MPI Status) :: status
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
 MPI_Mrecv(buf, count, datatype, message, status, ierror)
10
 TYPE(*), DIMENSION(..) :: buf
11
 INTEGER, INTENT(IN) :: count
12
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
 TYPE(MPI_Message), INTENT(INOUT) :: message
14
 TYPE(MPI_Status) :: status
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
 MPI_Probe(source, tag, comm, status, ierror)
17
 INTEGER, INTENT(IN) :: source, tag
18
 TYPE(MPI_Comm), INTENT(IN) :: comm
19
 TYPE(MPI_Status) :: status
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
23
 TYPE(*), DIMENSION(..) :: buf
24
 INTEGER, INTENT(IN) :: count, source, tag
25
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
 TYPE(MPI_Comm), INTENT(IN) :: comm
27
 TYPE(MPI_Status) :: status
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
 MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
30
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
31
 INTEGER, INTENT(IN) :: count, source, tag
32
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
 TYPE(MPI_Comm), INTENT(IN) :: comm
34
 TYPE(MPI_Request), INTENT(OUT) :: request
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Request_free(request, ierror)
38
 TYPE(MPI_Request), INTENT(INOUT) :: request
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
 MPI_Request_get_status(request, flag, status, ierror)
41
 TYPE(MPI_Request), INTENT(IN) :: request
42
 LOGICAL, INTENT(OUT) :: flag
43
 TYPE(MPI_Status) :: status
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
47
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
48
```

```
INTEGER, INTENT(IN) :: count, dest, tag
 1
 2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 3
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 4
 5
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
 6
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
 7
 INTEGER, INTENT(IN) :: count, dest, tag
 8
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 9
 TYPE(MPI_Comm), INTENT(IN) :: comm
 10
 TYPE(MPI_Request), INTENT(OUT) :: request
 11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 12
 13
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
 14
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
 15
 INTEGER, INTENT(IN) :: count, dest, tag
 16
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 17
 TYPE(MPI_Comm), INTENT(IN) :: comm
 18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 19
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
 20
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
 21
 INTEGER, INTENT(IN) :: count, dest, tag
 22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 23
 TYPE(MPI_Comm), INTENT(IN) :: comm
 24
 TYPE(MPI_Request), INTENT(OUT) :: request
 25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
 27
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
 28
 comm, status, ierror)
 29
 TYPE(*), DIMENSION(..) :: buf
 30
 INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
 31
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 32
 TYPE(MPI_Comm), INTENT(IN) :: comm
 33
 TYPE(MPI_Status) :: status
 34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 35
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
 36
 recvcount, recvtype, source, recvtag, comm, status, ierror)
 37
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 38
 TYPE(*), DIMENSION(..) :: recvbuf
 39
 INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
 40
 recvtag
 41
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 42
 TYPE(MPI_Comm), INTENT(IN) :: comm
 43
 TYPE(MPI_Status) :: status
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
 46
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
 47
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
 48
```

```
1
 INTEGER, INTENT(IN) :: count, dest, tag
2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
 TYPE(MPI_Comm), INTENT(IN) :: comm
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
6
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
7
 INTEGER, INTENT(IN) :: count, dest, tag
8
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
 TYPE(MPI_Comm), INTENT(IN) :: comm
10
 TYPE(MPI_Request), INTENT(OUT) :: request
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
 MPI_Startall(count, array_of_requests, ierror)
14
 INTEGER, INTENT(IN) :: count
15
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
 MPI_Start(request, ierror)
18
 TYPE(MPI_Request), INTENT(INOUT) :: request
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
22
 INTEGER, INTENT(IN) :: count
23
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
^{24}
 LOGICAL, INTENT(OUT) :: flag
25
 TYPE(MPI_Status) :: array_of_statuses(*)
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
 MPI_Testany(count, array_of_requests, index, flag, status, ierror)
28
 INTEGER, INTENT(IN) :: count
29
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
30
 INTEGER, INTENT(OUT) :: index
31
 LOGICAL, INTENT(OUT) :: flag
32
 TYPE(MPI_Status) :: status
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 MPI_Test_cancelled(status, flag, ierror)
36
 TYPE(MPI_Status), INTENT(IN) :: status
37
 LOGICAL, INTENT(OUT) :: flag
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
 MPI_Test(request, flag, status, ierror)
40
 TYPE(MPI_Request), INTENT(INOUT) :: request
41
 LOGICAL, INTENT(OUT) :: flag
42
 TYPE(MPI_Status) :: status
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
 MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
46
 array_of_statuses, ierror)
47
 INTEGER, INTENT(IN) :: incount
48
```

```
1
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
 2
 INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
 3
 TYPE(MPI_Status) :: array_of_statuses(*)
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 4
 5
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
 6
 INTEGER, INTENT(IN) :: count
 7
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
 8
 TYPE(MPI_Status) :: array_of_statuses(*)
 9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 10
 11
MPI_Waitany(count, array_of_requests, index, status, ierror)
 INTEGER, INTENT(IN) :: count
 12
 13
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
 14
 INTEGER, INTENT(OUT) :: index
 15
 TYPE(MPI_Status) :: status
 16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 17
MPI_Wait(request, status, ierror)
 18
 TYPE(MPI_Request), INTENT(INOUT) :: request
 19
 TYPE(MPI_Status) :: status
 20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 21
 22
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
 23
 array_of_statuses, ierror)
 24
 INTEGER, INTENT(IN) :: incount
 25
 TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
 26
 INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
 27
 TYPE(MPI_Status) :: array_of_statuses(*)
 28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
 30
A.4.2 Datatypes Fortran 2008 Bindings
 31
 32
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
 33
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
 34
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
 35
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2
 36
 37
MPI_Get_address(location, address, ierror)
 38
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: location
 39
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
 40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 41
MPI_Get_elements(status, datatype, count, ierror)
 42
 TYPE(MPI_Status), INTENT(IN) :: status
 43
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 44
 INTEGER, INTENT(OUT) :: count
 45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 46
 47
MPI_Get_elements_x(status, datatype, count, ierror)
 48
```

```
1
 TYPE(MPI_Status), INTENT(IN) :: status
2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
6
 position, ierror)
7
 CHARACTER(LEN=*), INTENT(IN) :: datarep
8
 TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
9
 TYPE(*), DIMENSION(..) :: outbuf
10
 INTEGER, INTENT(IN) :: incount
11
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
 MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
17
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
 INTEGER, INTENT(IN) :: incount
19
 CHARACTER(LEN=*), INTENT(IN) :: datarep
20
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
 MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
23
 TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
24
 TYPE(*), DIMENSION(..) :: outbuf
25
 INTEGER, INTENT(IN) :: incount, outsize
26
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
 INTEGER, INTENT(INOUT) :: position
28
 TYPE(MPI_Comm), INTENT(IN) :: comm
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
 MPI_Pack_size(incount, datatype, comm, size, ierror)
32
 INTEGER, INTENT(IN) :: incount
33
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
 TYPE(MPI_Comm), INTENT(IN) :: comm
35
 INTEGER, INTENT(OUT) :: size
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
 MPI_Type_commit(datatype, ierror)
38
 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Type_contiguous(count, oldtype, newtype, ierror)
42
 INTEGER, INTENT(IN) :: count
43
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
44
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
 MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
47
 array_of_distribs, array_of_dargs, array_of_psizes, order,
48
```

1 oldtype, newtype, ierror) 2 INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims), array_of_distribs(ndims), array_of_dargs(ndims), array_of_psizes(ndims), order 4 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 5 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 6 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements, 9 oldtype, newtype, ierror) 10 INTEGER, INTENT(IN) :: count, blocklength 11 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: 12array_of_displacements(count) 13 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 14TYPE(MPI_Datatype), INTENT(OUT) :: newtype 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617MPI_Type_create_hindexed(count, array_of_blocklengths, 18 array_of_displacements, oldtype, newtype, ierror) 19 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: 2021array_of_displacements(count) 22 TYPE(MPI_Datatype), INTENT(IN) :: oldtype TYPE(MPI_Datatype), INTENT(OUT) :: newtype 2324INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype, 26ierror) 27INTEGER, INTENT(IN) :: count, blocklength 28 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride 29 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 30 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 MPI_Type_create_indexed_block(count, blocklength, array_of_displacements, 34 oldtype, newtype, ierror) 35 INTEGER, INTENT(IN) :: count, blocklength, 36 array_of_displacements(count) 37 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 38 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) 41 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent 42TYPE(MPI_Datatype), INTENT(IN) :: oldtype 43 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546MPI_Type_create_struct(count, array_of_blocklengths, 47array_of_displacements, array_of_types, newtype, ierror) 48

```
1
 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
2
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
3
 array_of_displacements(count)
4
 TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
5
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
8
 array_of_starts, order, oldtype, newtype, ierror)
9
 INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
10
 array_of_subsizes(ndims), array_of_starts(ndims), order
11
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
12
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 MPI_Type_dup(oldtype, newtype, ierror)
16
 TYPE(MPI_Datatype), INTENT(IN) :: oldtype
17
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
 MPI_Type_free(datatype, ierror)
20
 TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
24
 array_of_integers, array_of_addresses, array_of_datatypes,
25
 ierror)
26
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
 INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
28
 INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
29
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
30
 array_of_addresses(max_addresses)
31
 TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
32
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
 MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
34
 combiner, ierror)
35
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
 INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
37
 combiner
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 MPI_Type_get_extent(datatype, lb, extent, ierror)
41
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
 MPI_Type_get_extent_x(datatype, lb, extent, ierror)
45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

1 MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) 2 TYPE(MPI_Datatype), INTENT(IN) :: datatype 3 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 5MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror) 6 TYPE(MPI_Datatype), INTENT(IN) :: datatype 7 INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, 11 oldtype, newtype, ierror) INTEGER, INTENT(IN) :: count, array_of_blocklengths(count), 12array_of_displacements(count) 13 14TYPE(MPI_Datatype), INTENT(IN) :: oldtype 15TYPE(MPI_Datatype), INTENT(OUT) :: newtype 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17MPI_Type_size(datatype, size, ierror) 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19 INTEGER, INTENT(OUT) :: size 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI_Type_size_x(datatype, size, ierror) 23TYPE(MPI_Datatype), INTENT(IN) :: datatype 24INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror) 27INTEGER, INTENT(IN) :: count, blocklength, stride 28 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 29 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount, 33 datatype, ierror) 34 CHARACTER(LEN=*), INTENT(IN) :: datarep 35 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf 36 TYPE(*), DIMENSION(..) :: outbuf 37 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize 38 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position 39 INTEGER, INTENT(IN) :: outcount 40TYPE(MPI_Datatype), INTENT(IN) :: datatype 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, 43 ierror) 44 TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf 45TYPE(*), DIMENSION(..) :: outbuf 46INTEGER, INTENT(IN) :: insize, outcount 47INTEGER, INTENT(INOUT) :: position 48

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5
 A.4.3 Collective Communication Fortran 2008 Bindings
6
7
 MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
8
 recvtype, comm, info, request, ierror)
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_Info), INTENT(IN) :: info
15
 TYPE(MPI_Request), INTENT(OUT) :: request
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
 MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
18
 comm, ierror)
19
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
20
 TYPE(*), DIMENSION(..) :: recvbuf
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
26
 MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
27
 displs, recvtype, comm, info, request, ierror)
28
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
29
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
30
 INTEGER, INTENT(IN) :: sendcount
31
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
32
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
33
 TYPE(MPI_Comm), INTENT(IN) :: comm
34
 TYPE(MPI_Info), INTENT(IN) :: info
35
 TYPE(MPI_Request), INTENT(OUT) :: request
36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
 MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
38
 recvtype, comm, ierror)
39
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
40
 TYPE(*), DIMENSION(..) :: recvbuf
41
 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
42
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
 TYPE(MPI_Comm), INTENT(IN) :: comm
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
47
 request, ierror)
48
```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 1 2 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 3 INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype 4 TYPE(MPI_Op), INTENT(IN) :: op 5TYPE(MPI_Comm), INTENT(IN) :: comm 6 7 TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) 11 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 12TYPE(*), DIMENSION(..) :: recvbuf 13 INTEGER, INTENT(IN) :: count 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15TYPE(MPI_Op), INTENT(IN) :: op 16TYPE(MPI_Comm), INTENT(IN) :: comm 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19 MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request, ierror) 2021TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 22 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 23INTEGER, INTENT(IN) :: sendcount, recvcount 24TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 25TYPE(MPI_Comm), INTENT(IN) :: comm 26TYPE(MPI_Info), INTENT(IN) :: info 27TYPE(MPI_Request), INTENT(OUT) :: request 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 30 comm, ierror) 31TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 32 TYPE(*), DIMENSION(..) :: recvbuf 33 INTEGER, INTENT(IN) :: sendcount, recvcount 34 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 35 TYPE(MPI_Comm), INTENT(IN) :: comm 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 38 MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf, 39 recvcounts, rdispls, recvtype, comm, info, request, ierror) 40TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 41 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 42INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), 43 recvcounts(*), rdispls(*) 44TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 45TYPE(MPI_Comm), INTENT(IN) :: comm 46TYPE(MPI_Info), INTENT(IN) :: info 47TYPE(MPI_Request), INTENT(OUT) :: request 48

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
3
 rdispls, recvtype, comm, ierror)
4
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
5
 TYPE(*), DIMENSION(..) :: recvbuf
6
 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
7
 rdispls(*)
8
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
 TYPE(MPI_Comm), INTENT(IN) :: comm
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
 MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
13
 recvcounts, rdispls, recvtypes, comm, info, request, ierror)
14
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
16
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
17
 recvcounts(*), rdispls(*)
18
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
19
 recvtypes(*)
20
 TYPE(MPI_Comm), INTENT(IN) :: comm
21
 TYPE(MPI_Info), INTENT(IN) :: info
22
 TYPE(MPI_Request), INTENT(OUT) :: request
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
25
 rdispls, recvtypes, comm, ierror)
26
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
27
 TYPE(*), DIMENSION(..) :: recvbuf
28
 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
29
 rdispls(*)
30
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
31
 TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
32
 TYPE(MPI_Comm), INTENT(IN) :: comm
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 MPI_Barrier(comm, ierror)
36
 TYPE(MPI_Comm), INTENT(IN) :: comm
37
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
 MPI_Barrier_init(comm, info, request, ierror)
39
 TYPE(MPI_Comm), INTENT(IN) :: comm
40
 TYPE(MPI_Info), INTENT(IN) :: info
41
 TYPE(MPI_Request), INTENT(OUT) :: request
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
 MPI_Bcast(buffer, count, datatype, root, comm, ierror)
45
 TYPE(*), DIMENSION(..) :: buffer
46
 INTEGER, INTENT(IN) :: count, root
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
 TYPE(MPI_Comm), INTENT(IN) :: comm
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
 5
 INTEGER, INTENT(IN) :: count, root
 6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 7
 TYPE(MPI_Comm), INTENT(IN) :: comm
 8
 TYPE(MPI_Info), INTENT(IN) :: info
 9
 TYPE(MPI_Request), INTENT(OUT) :: request
 10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 11
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
 12
 13
 ierror)
 14
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 15
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
 16
 INTEGER, INTENT(IN) :: count
 17
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 18
 TYPE(MPI_Op), INTENT(IN) :: op
 19
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Info), INTENT(IN) :: info
 20
 21
 TYPE(MPI_Request), INTENT(OUT) :: request
 22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 23
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
 24
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 25
 TYPE(*), DIMENSION(..) :: recvbuf
 26
 INTEGER, INTENT(IN) :: count
 27
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 28
 TYPE(MPI_Op), INTENT(IN) :: op
 29
 TYPE(MPI_Comm), INTENT(IN) :: comm
 30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 31
 32
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 33
 root, comm, info, request, ierror)
 34
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 35
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 36
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
 37
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 38
 TYPE(MPI_Comm), INTENT(IN) :: comm
 39
 TYPE(MPI_Info), INTENT(IN) :: info
 40
 TYPE(MPI_Request), INTENT(OUT) :: request
 41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 42
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 43
 root, comm, ierror)
 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_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
4
 recvtype, root, comm, info, request, ierror)
5
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
6
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7
 INTEGER, INTENT(IN) :: sendcount, root
8
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
9
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
 TYPE(MPI_Comm), INTENT(IN) :: comm
11
 TYPE(MPI_Info), INTENT(IN) :: info
12
 TYPE(MPI_Request), INTENT(OUT) :: request
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
16
 recvtype, root, comm, ierror)
17
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
18
 TYPE(*), DIMENSION(..) :: recvbuf
19
 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
20
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
 TYPE(MPI_Comm), INTENT(IN) :: comm
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
 MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
24
 comm, request, ierror)
25
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
 INTEGER, INTENT(IN) :: sendcount, recvcount
28
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 TYPE(MPI_Request), INTENT(OUT) :: request
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
 MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
34
 recvtype, comm, request, ierror)
35
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
36
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
37
 INTEGER, INTENT(IN) :: sendcount
38
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
39
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
 TYPE(MPI_Comm), INTENT(IN) :: comm
41
 TYPE(MPI_Request), INTENT(OUT) :: request
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
 MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
44
 ierror)
45
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
 INTEGER, INTENT(IN) :: count
48
```

```
1
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 2
 TYPE(MPI_Op), INTENT(IN) :: op
 3
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Request), INTENT(OUT) :: request
 4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 5
 6
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 7
 comm, request, ierror)
 8
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 9
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 10
 INTEGER, INTENT(IN) :: sendcount, recvcount
 11
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 12
 TYPE(MPI_Comm), INTENT(IN) :: comm
 13
 TYPE(MPI_Request), INTENT(OUT) :: request
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
 17
 rdispls, recvtype, comm, request, ierror)
 18
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 19
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 20
 21
 recvcounts(*), rdispls(*)
 22
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 23
 TYPE(MPI_Comm), INTENT(IN) :: comm
 24
 TYPE(MPI_Request), INTENT(OUT) :: request
 25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
 27
 recvcounts, rdispls, recvtypes, comm, request, ierror)
 28
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 29
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 30
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 31
 recvcounts(*), rdispls(*)
 32
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 33
 recvtypes(*)
 34
 TYPE(MPI_Comm), INTENT(IN) :: comm
 35
 TYPE(MPI_Request), INTENT(OUT) :: request
 36
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 37
 38
MPI_Ibarrier(comm, request, ierror)
 39
 TYPE(MPI_Comm), INTENT(IN) :: comm
 40
 TYPE(MPI_Request), INTENT(OUT) :: request
 41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 42
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
 43
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
 44
 INTEGER, INTENT(IN) :: count, root
 45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 46
 TYPE(MPI_Comm), INTENT(IN) :: comm
 47
 TYPE(MPI_Request), INTENT(OUT) :: request
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
3
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
5
 INTEGER, 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
12
 MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
13
 root, comm, request, ierror)
14
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
16
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
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_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
22
 recvtype, root, comm, request, ierror)
23
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
 INTEGER, INTENT(IN) :: sendcount, root
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_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
33
 request, ierror)
34
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
36
 INTEGER, INTENT(IN) :: recvcount
37
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
 TYPE(MPI_Op), INTENT(IN) :: op
39
 TYPE(MPI_Comm), INTENT(IN) :: comm
40
 TYPE(MPI_Request), INTENT(OUT) :: request
41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
 MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
43
 request, ierror)
44
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
 sendbuf
45
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
46
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
 TYPE(MPI_Op), INTENT(IN) :: op
 2
 TYPE(MPI_Comm), INTENT(IN) :: comm
 3
 TYPE(MPI_Request), INTENT(OUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 4
 5
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
 6
 ierror)
 7
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 8
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 9
 INTEGER, INTENT(IN) :: count, root
 10
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 11
 TYPE(MPI_Op), INTENT(IN) :: op
 12
 TYPE(MPI_Comm), INTENT(IN) :: comm
 13
 TYPE(MPI_Request), INTENT(OUT) :: request
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
 17
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 18
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 19
 INTEGER, INTENT(IN) :: count
 20
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 21
 TYPE(MPI_Op), INTENT(IN) :: op
 22
 TYPE(MPI_Comm), INTENT(IN) :: comm
 23
 TYPE(MPI_Request), INTENT(OUT) :: request
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
 26
 root, comm, request, ierror)
 27
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 28
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 29
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
 30
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 31
 TYPE(MPI_Comm), INTENT(IN) :: comm
 32
 TYPE(MPI_Request), INTENT(OUT) :: request
 33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 34
 35
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
 36
 recvtype, root, comm, request, ierror)
 37
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 38
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 39
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
 40
 INTEGER, INTENT(IN) :: recvcount, root
 41
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 42
 TYPE(MPI_Comm), INTENT(IN) :: comm
 43
 TYPE(MPI_Request), INTENT(OUT) :: request
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
MPI_Op_commutative(op, commute, ierror)
 46
 TYPE(MPI_Op), INTENT(IN) :: op
 47
 LOGICAL, INTENT(OUT) :: commute
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Op_create(user_fn, commute, op, ierror)
3
 PROCEDURE(MPI_User_function) :: user_fn
4
 LOGICAL, INTENT(IN) :: commute
5
 TYPE(MPI_Op), INTENT(OUT) :: op
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
 MPI_Op_free(op, ierror)
9
 TYPE(MPI_Op), INTENT(INOUT) :: op
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
 MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
12
 request, ierror)
13
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
14
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
15
 INTEGER, INTENT(IN) :: count, root
16
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
 TYPE(MPI_Op), INTENT(IN) :: op
18
 TYPE(MPI_Comm), INTENT(IN) :: comm
19
 TYPE(MPI_Info), INTENT(IN) :: info
20
 TYPE(MPI_Request), INTENT(OUT) :: request
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
24
 TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
25
 TYPE(*), DIMENSION(..) :: inoutbuf
26
 INTEGER, INTENT(IN) :: count
27
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
 TYPE(MPI_Op), INTENT(IN) :: op
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
 MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
^{31}
 comm, info, request, ierror)
32
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
33
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
34
 INTEGER, INTENT(IN) :: recvcount
35
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
 TYPE(MPI_Op), INTENT(IN) :: op
37
 TYPE(MPI_Comm), INTENT(IN) :: comm
38
 TYPE(MPI_Info), INTENT(IN) :: info
39
 TYPE(MPI_Request), INTENT(OUT) :: request
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
 MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
43
 ierror)
44
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
45
 TYPE(*), DIMENSION(..) :: recvbuf
46
 INTEGER, INTENT(IN) :: recvcount
47
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
 TYPE(MPI_Op), INTENT(IN) :: op
 TYPE(MPI_Comm), INTENT(IN) :: comm
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
 5
 info, request, ierror)
 6
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 7
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 8
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
 9
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 10
 TYPE(MPI_Op), INTENT(IN) :: op
 11
 TYPE(MPI_Comm), INTENT(IN) :: comm
 12
 TYPE(MPI_Info), INTENT(IN) :: info
 13
 TYPE(MPI_Request), INTENT(OUT) :: request
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
 17
 ierror)
 18
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 19
 TYPE(*), DIMENSION(..) :: recvbuf
 INTEGER, INTENT(IN) :: recvcounts(*)
 20
 21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 22
 TYPE(MPI_Op), INTENT(IN) :: op
 23
 TYPE(MPI_Comm), INTENT(IN) :: comm
 24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 25
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
 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_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
 35
 ierror)
 36
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 37
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 38
 INTEGER, INTENT(IN) :: count
 39
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 40
 TYPE(MPI_Op), INTENT(IN) :: op
 41
 TYPE(MPI_Comm), INTENT(IN) :: comm
 42
 TYPE(MPI_Info), INTENT(IN) :: info
 43
 TYPE(MPI_Request), INTENT(OUT) :: request
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
 46
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 47
 TYPE(*), DIMENSION(..) :: recvbuf
 48
```

```
1
 INTEGER, INTENT(IN) :: count
2
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
 TYPE(MPI_Op), INTENT(IN) :: op
4
 TYPE(MPI_Comm), INTENT(IN) :: comm
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
7
 recvtype, root, comm, info, request, ierror)
8
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
11
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
 TYPE(MPI_Comm), INTENT(IN) :: comm
13
 TYPE(MPI_Info), INTENT(IN) ::
 info
14
 TYPE(MPI_Request), INTENT(OUT) :: request
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
18
 root, comm, ierror)
19
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
20
 TYPE(*), DIMENSION(..) :: recvbuf
21
 INTEGER, INTENT(IN) :: sendcount, recvcount, root
22
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23
 TYPE(MPI_Comm), INTENT(IN) :: comm
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
 MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
26
 recvcount, recvtype, root, comm, info, request, ierror)
27
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
28
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
29
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
30
 INTEGER, INTENT(IN) :: recvcount, root
31
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32
 TYPE(MPI_Comm), INTENT(IN) :: comm
33
 TYPE(MPI_Info), INTENT(IN) :: info
34
 TYPE(MPI_Request), INTENT(OUT) :: request
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
38
 recvtype, root, comm, ierror)
39
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
40
 TYPE(*), DIMENSION(..) :: recvbuf
41
 INTEGER, INTENT(IN) :: sendcounts(*), displs(*), 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
47
48
```

A.4.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings 1 2 MPI_Comm_compare(comm1, comm2, result, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2 4 INTEGER, INTENT(OUT) :: result 5INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 7 MPI_Comm_create(comm, group, newcomm, ierror) 8 TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Group), INTENT(IN) :: group 9 10 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12MPI_Comm_create_group(comm, group, tag, newcomm, ierror) 13 TYPE(MPI_Comm), INTENT(IN) :: comm 14TYPE(MPI_Group), INTENT(IN) :: group 15INTEGER, INTENT(IN) :: tag 16TYPE(MPI_Comm), INTENT(OUT) :: newcomm 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19 MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, 20extra_state, ierror) 21PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn 22 PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn 23INTEGER, INTENT(OUT) :: comm_keyval 24INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI_Comm_delete_attr(comm, comm_keyval, ierror) 27TYPE(MPI_Comm), INTENT(IN) :: comm 28 INTEGER, INTENT(IN) :: comm_keyval 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31MPI_Comm_dup(comm, newcomm, ierror) 32 TYPE(MPI_Comm), INTENT(IN) :: comm 33 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in, 36 attribute_val_out, flag, ierror) 37 TYPE(MPI_Comm) :: oldcomm 38 INTEGER :: comm_keyval 39 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in 40 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out 41 LOGICAL :: flag 42INTEGER :: ierror 43 44MPI_Comm_dup_with_info(comm, info, newcomm, ierror) 45TYPE(MPI_Comm), INTENT(IN) :: comm 46TYPE(MPI_Info), INTENT(IN) :: info 47TYPE(MPI_Comm), INTENT(OUT) :: newcomm 48

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
 MPI_Comm_free(comm, ierror)
3
 TYPE(MPI_Comm), INTENT(INOUT) :: comm
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
 MPI_Comm_free_keyval(comm_keyval, ierror)
7
 INTEGER, INTENT(INOUT) :: comm_kevval
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
 MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
10
 TYPE(MPI_Comm), INTENT(IN) :: comm
11
 INTEGER, INTENT(IN) :: comm_keyval
12
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
13
 LOGICAL, INTENT(OUT) :: flag
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
 MPI_Comm_get_info(comm, info_used, ierror)
17
 TYPE(MPI_Comm), INTENT(IN) :: comm
18
 TYPE(MPI_Info), INTENT(OUT) :: info_used
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
21
 TYPE(MPI_Comm), INTENT(IN) :: comm
22
 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
23
 INTEGER, INTENT(OUT) :: resultlen
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_Comm_group(comm, group, ierror)
27
 TYPE(MPI_Comm), INTENT(IN) :: comm
28
 TYPE(MPI_Group), INTENT(OUT) :: group
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
 MPI_Comm_idup(comm, newcomm, request, ierror)
^{31}
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
33
 TYPE(MPI_Request), INTENT(OUT) :: request
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
37
 TYPE(MPI_Comm), INTENT(IN) :: comm
38
 TYPE(MPI_Info), INTENT(IN) :: info
39
 TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
40
 TYPE(MPI_Request), INTENT(OUT) :: request
41
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
 MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
43
 attribute_val_out, flag, ierror)
44
 TYPE(MPI_Comm) :: oldcomm
45
 INTEGER :: comm_keyval
46
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
47
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
48
```

LOGICAL :: flag INTEGER :: ierror	1 2
MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,	3 4
ierror) TYPE(MPI_Comm) :: comm	5 6
INTEGER :: comm_keyval INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state INTEGER :: ierror	7 8 9
MPI_Comm_rank(comm, rank, ierror)	10
TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(OUT) :: rank	11 12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13 14
<pre>MPI_Comm_remote_group(comm, group, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	14 15 16
TYPE(MPI_Group), INTENT(OUT) :: group INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
	18 19
<pre>MPI_Comm_remote_size(comm, size, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	20
INTEGER, INTENT(OUT) :: size	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)	23 24
TYPE(MPI_Comm), INTENT(IN) :: comm	25
INTEGER, INTENT(IN) :: comm_keyval	26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
MPI_Comm_set_info(comm, info, ierror)	29
TYPE(MPI_Comm), INTENT(IN) :: comm	30 31
TYPE(MPI_Info), INTENT(IN) :: info	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
MPI_Comm_set_name(comm, comm_name, ierror)	34
TYPE(MPI_Comm), INTENT(IN) :: comm	35
CHARACTER(LEN=*), INTENT(IN) :: comm_name	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MPI_Comm_size(comm, size, ierror)	38
TYPE(MPI_Comm), INTENT(IN) :: comm	39
INTEGER, INTENT(OUT) :: size	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41 42
MPI_Comm_split(comm, color, key, newcomm, ierror)	42
TYPE(MPI_Comm), INTENT(IN) :: comm	40
INTEGER, INTENT(IN) :: color, key	45
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
	48

```
1
 MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
\mathbf{2}
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 INTEGER, INTENT(IN) :: split_type, key
4
 TYPE(MPI_Info), INTENT(IN) :: info
5
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Comm_test_inter(comm, flag, ierror)
8
 TYPE(MPI_Comm), INTENT(IN) :: comm
9
 LOGICAL, INTENT(OUT) :: flag
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
 MPI_Group_compare(group1, group2, result, ierror)
13
 TYPE(MPI_Group), INTENT(IN) :: group1, group2
14
 INTEGER, INTENT(OUT) :: result
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
 MPI_Group_difference(group1, group2, newgroup, ierror)
17
 TYPE(MPI_Group), INTENT(IN) :: group1, group2
18
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
 MPI_Group_excl(group, n, ranks, newgroup, ierror)
22
 TYPE(MPI_Group), INTENT(IN) :: group
23
 INTEGER, INTENT(IN) :: n, ranks(n)
24
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Group_free(group, ierror)
27
 TYPE(MPI_Group), INTENT(INOUT) :: group
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 MPI_Group_incl(group, n, ranks, newgroup, ierror)
^{31}
 TYPE(MPI_Group), INTENT(IN) :: group
32
 INTEGER, INTENT(IN) :: n, ranks(n)
33
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
 MPI_Group_intersection(group1, group2, newgroup, ierror)
36
 TYPE(MPI_Group), INTENT(IN) :: group1, group2
37
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
 MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
41
 TYPE(MPI_Group), INTENT(IN) :: group
42
 INTEGER, INTENT(IN) :: n, ranges(3,n)
43
 TYPE(MPI_Group), INTENT(OUT) :: newgroup
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
 MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
46
 TYPE(MPI_Group), INTENT(IN) :: group
47
 INTEGER, INTENT(IN) :: n, ranges(3,n)
48
```

TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
<pre>MPI_Group_rank(group, rank, ierror)     TYPE(MPI_Group), INTENT(IN) :: group</pre>	3 4
INTEGER, INTENT(OUT) :: rank INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5 6 7
<pre>MPI_Group_size(group, size, ierror)     TYPE(MPI_Group), INTENT(IN) :: group</pre>	8
INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10 11
<pre>MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)     TYPE(MPI_Group), INTENT(IN) :: group1, group2</pre>	12 13
INTEGER, INTENT(IN) :: n, ranks1(n) INTEGER, INTENT(OUT) :: ranks2(n)	14 15 16
<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Group_union(group1, group2, newgroup, ierror)</pre>	10 17 18
TYPE(MPI_Group), INTENT(IN) :: group1, group2 TYPE(MPI_Group), INTENT(OUT) :: newgroup	19 20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21 22
<pre>MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,</pre>	23 24
INTEGER, INTENT(IN) :: local_leader, remote_leader, tag TYPE(MPI_Comm), INTENT(OUT) :: newintercomm	25 26 27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28 29
<pre>MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)     TYPE(MPI_Comm), INTENT(IN) :: intercomm     LOGICAL, INTENT(IN) :: high</pre>	30 31
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 33
<pre>MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,</pre>	34 35
extra_state, ierror) PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn	36 37
<pre>PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn INTEGER, INTENT(OUT) :: type_keyval INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state</pre>	38 39 40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40 41 42
<pre>MPI_Type_delete_attr(datatype, type_keyval, ierror)     TYPE(MPI_Datatype), INTENT(IN) :: datatype     INTEGER, INTENT(IN) :: type_keyval</pre>	43 44 45
<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,</pre>	46 47
attribute_val_out, flag, ierror)	48

```
1
 TYPE(MPI_Datatype) :: oldtype
2
 INTEGER :: type_keyval
3
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
4
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
5
 LOGICAL :: flag
6
 INTEGER :: ierror
7
 MPI_Type_free_keyval(type_keyval, ierror)
8
 INTEGER, INTENT(INOUT) :: type_keyval
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
12
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
 INTEGER, INTENT(IN) :: type_keyval
14
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
15
 LOGICAL, INTENT(OUT) :: flag
16
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
 MPI_Type_get_name(datatype, type_name, resultlen, ierror)
18
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
20
 INTEGER, INTENT(OUT) :: resultlen
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
24
 attribute_val_out, flag, ierror)
25
 TYPE(MPI_Datatype) :: oldtype
26
 INTEGER :: type_keyval
27
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
28
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
29
 LOGICAL :: flag
30
 INTEGER :: ierror
31
 MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
32
 ierror)
33
 TYPE(MPI_Datatype) :: datatype
34
 INTEGER :: type_keyval
35
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
36
 INTEGER, INTENT(OUT) :: ierror
37
38
 MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
39
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
 INTEGER, INTENT(IN) :: type_keyval
41
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
 MPI_Type_set_name(datatype, type_name, ierror)
44
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
 CHARACTER(LEN=*), INTENT(IN) :: type_name
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
 2
 extra_state, ierror)
 3
 PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
 4
 PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
 INTEGER, INTENT(OUT) :: win_keyval
 5
 6
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
 7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 8
MPI_Win_delete_attr(win, win_keyval, ierror)
 9
 TYPE(MPI_Win), INTENT(IN) :: win
 10
 INTEGER, INTENT(IN) :: win_keyval
 11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 12
 13
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
 14
 attribute_val_out, flag, ierror)
 15
 TYPE(MPI_Win) :: oldwin
 16
 INTEGER :: win_keyval
 17
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
 18
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
 19
 LOGICAL :: flag
 20
 INTEGER :: ierror
 21
MPI_Win_free_keyval(win_keyval, ierror)
 22
 INTEGER, INTENT(INOUT) :: win_keyval
 23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 24
 25
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
 26
 TYPE(MPI_Win), INTENT(IN) :: win
 27
 INTEGER, INTENT(IN) :: win_keyval
 28
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
 29
 LOGICAL, INTENT(OUT) :: flag
 30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 31
MPI_Win_get_name(win, win_name, resultlen, ierror)
 32
 TYPE(MPI_Win), INTENT(IN) :: win
 33
 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
 34
 INTEGER, INTENT(OUT) :: resultlen
 35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
 37
MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
 38
 attribute_val_out, flag, ierror)
 39
 TYPE(MPI_Win) :: oldwin
 40
 INTEGER :: win_keyval
 41
 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
 42
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
 43
 LOGICAL :: flag
 44
 INTEGER :: ierror
 45
MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
 46
 TYPE(MPI_Win) :: win
 47
 INTEGER :: win_keyval
 48
```

```
1
 INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
\mathbf{2}
 INTEGER :: ierror
3
 MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
4
 TYPE(MPI_Win), INTENT(IN) :: win
5
 INTEGER, INTENT(IN) :: win_keyval
6
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
 MPI_Win_set_name(win, win_name, ierror)
10
 TYPE(MPI_Win), INTENT(IN) :: win
11
 CHARACTER(LEN=*), INTENT(IN) :: win_name
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 A.4.5 Process Topologies Fortran 2008 Bindings
15
16
 MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
17
 TYPE(MPI_Comm), INTENT(IN) :: comm
18
 INTEGER, INTENT(IN) :: rank, maxdims
19
 INTEGER, INTENT(OUT) :: coords(maxdims)
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
 MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
22
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
23
 INTEGER, INTENT(IN) :: ndims, dims(ndims)
24
 LOGICAL, INTENT(IN) :: periods(ndims), reorder
25
 TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
 MPI_Cartdim_get(comm, ndims, ierror)
29
 TYPE(MPI_Comm), INTENT(IN) :: comm
30
 INTEGER, INTENT(OUT) :: ndims
31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
 MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
33
34
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: maxdims
35
 INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
36
37
 LOGICAL, INTENT(OUT) :: periods(maxdims)
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
 MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
40
 TYPE(MPI_Comm), INTENT(IN) :: comm
41
 INTEGER, INTENT(IN) :: ndims, dims(ndims)
42
 LOGICAL, INTENT(IN) :: periods(ndims)
43
 INTEGER, INTENT(OUT) :: newrank
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_Cart_rank(comm, coords, rank, ierror)
47
 TYPE(MPI_Comm), INTENT(IN) :: comm
48
 INTEGER, INTENT(IN) :: coords(*)
```

```
1
 INTEGER, INTENT(OUT) :: rank
 2
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
 4
 TYPE(MPI_Comm), INTENT(IN) :: comm
 5
 INTEGER, INTENT(IN) :: direction, disp
 6
 INTEGER, INTENT(OUT) :: rank_source, rank_dest
 7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 8
 9
MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
 10
 TYPE(MPI_Comm), INTENT(IN) :: comm
 11
 LOGICAL, INTENT(IN) :: remain_dims(*)
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
 12
 13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 14
MPI_Dims_create(nnodes, ndims, dims, ierror)
 15
 INTEGER, INTENT(IN) :: nnodes, ndims
 16
 INTEGER, INTENT(INOUT) :: dims(ndims)
 17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 18
 19
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
 20
 outdegree, destinations, destweights, info, reorder,
 21
 comm_dist_graph, ierror)
 22
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
 23
 INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
 24
 destinations(outdegree)
 25
 INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
 26
 TYPE(MPI_Info), INTENT(IN) :: info
 27
 LOGICAL, INTENT(IN) :: reorder
 28
 TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
 29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 30
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
 31
 info, reorder, comm_dist_graph, ierror)
 32
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
 33
 INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
 34
 INTEGER, INTENT(IN) :: weights(*)
 35
 TYPE(MPI_Info), INTENT(IN) :: info
 36
 LOGICAL, INTENT(IN) :: reorder
 37
 TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
 38
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 39
 40
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
 41
 maxoutdegree, destinations, destweights, ierror)
 42
 TYPE(MPI_Comm), INTENT(IN) :: comm
 43
 INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
 44
 INTEGER, INTENT(OUT) :: sources(maxindegree),
 45
 destinations(maxoutdegree)
 46
 INTEGER :: sourceweights(*), destweights(*)
 47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 48
```

```
1
 MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
\mathbf{2}
 TYPE(MPI_Comm), INTENT(IN) :: comm
3
 INTEGER, INTENT(OUT) :: indegree, outdegree
4
 LOGICAL, INTENT(OUT) :: weighted
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
 MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
7
 ierror)
8
 TYPE(MPI_Comm), INTENT(IN) :: comm_old
9
 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
10
 LOGICAL, INTENT(IN) :: reorder
11
 TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
 MPI_Graphdims_get(comm, nnodes, nedges, ierror)
15
 TYPE(MPI_Comm), INTENT(IN) :: comm
16
 INTEGER, INTENT(OUT) :: nnodes, nedges
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
 MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
19
 TYPE(MPI_Comm), INTENT(IN) :: comm
20
 INTEGER, INTENT(IN) :: maxindex, maxedges
21
 INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
22
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
 MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
25
 TYPE(MPI_Comm), INTENT(IN) :: comm
26
 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
27
 INTEGER, INTENT(OUT) :: newrank
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
 MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
30
 TYPE(MPI_Comm), INTENT(IN) :: comm
31
 INTEGER, INTENT(IN) :: rank, maxneighbors
32
 INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
36
 TYPE(MPI_Comm), INTENT(IN) :: comm
37
 INTEGER, INTENT(IN) :: rank
38
 INTEGER, INTENT(OUT) :: nneighbors
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
 MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
41
 recvtype, comm, request, ierror)
42
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
43
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
 INTEGER, INTENT(IN) :: sendcount, recvcount
45
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46
 TYPE(MPI_Comm), INTENT(IN) :: comm
47
 TYPE(MPI_Request), INTENT(OUT) :: request
48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 2
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
 3
 displs, recvtype, comm, request, ierror)
 4
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 5
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
 6
 INTEGER, INTENT(IN) :: sendcount
 7
 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
 8
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 9
 TYPE(MPI_Comm), INTENT(IN) :: comm
 10
 TYPE(MPI_Request), INTENT(OUT) :: request
 11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 12
 13
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
 14
 recvtype, comm, request, ierror)
 15
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 16
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 17
 INTEGER, INTENT(IN) :: sendcount, recvcount
 18
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 19
 TYPE(MPI_Comm), INTENT(IN) :: comm
 TYPE(MPI_Request), INTENT(OUT) :: request
 20
 21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 22
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 23
 recvcounts, rdispls, recvtype, comm, request, ierror)
 24
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 25
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 26
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 27
 recvcounts(*), rdispls(*)
 28
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 29
 TYPE(MPI_Comm), INTENT(IN) :: comm
 30
 TYPE(MPI_Request), INTENT(OUT) :: request
 31
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 32
 33
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
 34
 recvcounts, rdispls, recvtypes, comm, request, ierror)
 35
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 36
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 37
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
 38
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
 39
 sdispls(*), rdispls(*)
 40
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 41
 recvtypes(*)
 42
 TYPE(MPI_Comm), INTENT(IN) :: comm
 43
 TYPE(MPI_Request), INTENT(OUT) :: request
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
 46
 recvcount, recvtype, comm, info, request, ierror)
 47
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 48
```

```
1
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
2
 INTEGER, INTENT(IN) :: sendcount, recvcount
3
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
 TYPE(MPI_Comm), INTENT(IN) :: comm
5
 TYPE(MPI_Info), INTENT(IN) :: info
6
 TYPE(MPI_Request), INTENT(OUT) :: request
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
 MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
9
 recvtype, comm, ierror)
10
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
11
 TYPE(*), DIMENSION(..) :: recvbuf
12
 INTEGER, INTENT(IN) :: sendcount, recvcount
13
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14
 TYPE(MPI_Comm), INTENT(IN) :: comm
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
18
 recvcounts, displs, recvtype, comm, info, request, ierror)
19
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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_Info), INTENT(IN) :: info
26
 TYPE(MPI_Request), INTENT(OUT) :: request
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
 MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
29
 displs, recvtype, comm, ierror)
30
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
31
 TYPE(*), DIMENSION(..) :: recvbuf
32
 INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
33
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
 TYPE(MPI_Comm), INTENT(IN) :: comm
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
38
 recvcount, recvtype, comm, info, request, ierror)
39
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
40
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
41
 INTEGER, INTENT(IN) :: sendcount, recvcount
42
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
 TYPE(MPI_Comm), INTENT(IN) :: comm
44
 TYPE(MPI_Info), INTENT(IN) :: info
45
 TYPE(MPI_Request), INTENT(OUT) :: request
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
48
```

```
1
 recvtype, comm, ierror)
 2
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 3
 TYPE(*), DIMENSION(..) :: recvbuf
 INTEGER, INTENT(IN) :: sendcount, recvcount
 4
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 5
 6
 TYPE(MPI_Comm), INTENT(IN) :: comm
 7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 8
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
 9
 recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
 10
 ierror)
 11
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
 12
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 13
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
 14
 recvcounts(*), rdispls(*)
 15
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 16
 TYPE(MPI_Comm), INTENT(IN) :: comm
 17
 TYPE(MPI_Info), INTENT(IN) :: info
 18
 TYPE(MPI_Request), INTENT(OUT) :: request
 19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
 21
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
 22
 recvcounts, rdispls, recvtype, comm, ierror)
 23
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 24
 TYPE(*), DIMENSION(..) :: recvbuf
 25
 INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
 26
 rdispls(*)
 27
 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
 28
 TYPE(MPI_Comm), INTENT(IN) :: comm
 29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 30
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
 31
 recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
 32
 ierror)
 33
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
 34
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
 35
 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
 36
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
 37
 sdispls(*), rdispls(*)
 38
 TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
 39
 recvtypes(*)
 40
 TYPE(MPI_Comm), INTENT(IN) :: comm
 41
 TYPE(MPI_Info), INTENT(IN) :: info
 42
 TYPE(MPI_Request), INTENT(OUT) :: request
 43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 44
 45
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
 46
 recvcounts, rdispls, recvtypes, comm, ierror)
 47
 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
 48
```

```
1
 TYPE(*), DIMENSION(..) :: recvbuf
2
 INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
3
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
4
 TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
5
 TYPE(MPI_Comm), INTENT(IN) :: comm
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Topo_test(comm, status, ierror)
8
 TYPE(MPI_Comm), INTENT(IN) :: comm
9
 INTEGER, INTENT(OUT) :: status
10
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
13
 A.4.6 MPI Environmental Management Fortran 2008 Bindings
14
 DOUBLE PRECISION MPI_Wtick()
15
16
 DOUBLE PRECISION MPI_Wtime()
17
 MPI_Abort(comm, errorcode, ierror)
18
 TYPE(MPI_Comm), INTENT(IN) :: comm
19
 INTEGER, INTENT(IN) :: errorcode
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Add_error_class(errorclass, ierror)
23
 INTEGER, INTENT(OUT) :: errorclass
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_Add_error_code(errorclass, errorcode, ierror)
 INTEGER, INTENT(IN) :: errorclass
27
 INTEGER, INTENT(OUT) :: errorcode
28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
 MPI_Add_error_string(errorcode, string, ierror)
^{31}
 INTEGER, INTENT(IN) :: errorcode
32
 CHARACTER(LEN=*), INTENT(IN) :: string
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 MPI_Alloc_mem(size, info, baseptr, ierror)
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
36
37
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
38
 TYPE(MPI_Info), INTENT(IN) :: info
 TYPE(C_PTR), INTENT(OUT) :: baseptr
39
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
 MPI_Comm_call_errhandler(comm, errorcode, ierror)
42
 TYPE(MPI_Comm), INTENT(IN) :: comm
43
 INTEGER, INTENT(IN) :: errorcode
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
47
 PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
48
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Comm_get_errhandler(comm, errhandler, ierror)	2
TYPE(MPI_Comm), INTENT(IN) :: comm	3 4
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	4 5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MPI_Comm_set_errhandler(comm, errhandler, ierror)	7
TYPE(MPI_Comm), INTENT(IN) :: comm	8
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MDI Employ dian free (amplendian i amper)	11
<pre>MPI_Errhandler_free(errhandler, ierror)     TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler</pre>	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
	14 15
MPI_Error_class(errorcode, errorclass, ierror)	15
INTEGER, INTENT(IN) :: errorcode	17
INTEGER, INTENT(OUT) :: errorclass INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
	19
MPI_Error_string(errorcode, string, resultlen, ierror)	20
INTEGER, INTENT(IN) :: errorcode	21
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string	22
INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
INTEGER, OFFICINAL, INTENT(COT) TETTOT	24
MPI_File_call_errhandler(fh, errorcode, ierror)	25 26
TYPE(MPI_File), INTENT(IN) :: fh	20
INTEGER, INTENT(IN) :: errorcode	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
<pre>MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)</pre>	30
<pre>PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn</pre>	31
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
MPI_File_get_errhandler(file, errhandler, ierror)	34
TYPE(MPI_File), INTENT(IN) :: file	35
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	36 37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
MPI_File_set_errhandler(file, errhandler, ierror)	39
TYPE(MPI_File), INTENT(IN) :: file	40
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MPI_Finalized(flag, ierror)	43
LOGICAL, INTENT(OUT) :: flag	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
MPI_Finalize(ierror)	46 47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
 MPI_Free_mem(base, ierror)
\mathbf{2}
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
3
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
 MPI_Get_library_version(version, resultlen, ierror)
5
 CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
6
 INTEGER, INTENT(OUT) :: resultlen
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
 MPI_Get_processor_name(name, resultlen, ierror)
10
 CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
11
 INTEGER, INTENT(OUT) :: resultlen
12
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
 MPI_Get_version(version, subversion, ierror)
14
 INTEGER, INTENT(OUT) :: version, subversion
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_Initialized(flag, ierror)
18
 LOGICAL, INTENT(OUT) :: flag
19
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
 MPI_Init(ierror)
21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
 MPI_Win_call_errhandler(win, errorcode, ierror)
^{24}
 TYPE(MPI_Win), INTENT(IN) :: win
25
 INTEGER, INTENT(IN) :: errorcode
26
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
 MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
28
 PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
29
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
 MPI_Win_get_errhandler(win, errhandler, ierror)
33
 TYPE(MPI_Win), INTENT(IN) :: win
34
 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
 MPI_Win_set_errhandler(win, errhandler, ierror)
37
 TYPE(MPI_Win), INTENT(IN) :: win
38
 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
42
 A.4.7 The Info Object Fortran 2008 Bindings
43
44
 MPI_Info_create(info, ierror)
45
 TYPE(MPI_Info), INTENT(OUT) :: info
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_Info_delete(info, key, ierror)
48
```

1 TYPE(MPI_Info), INTENT(IN) :: info 2 CHARACTER(LEN=*), INTENT(IN) :: key 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI_Info_dup(info, newinfo, ierror) 5TYPE(MPI_Info), INTENT(IN) :: info 6 TYPE(MPI_Info), INTENT(OUT) :: newinfo 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI_Info_free(info, ierror) 10 TYPE(MPI_Info), INTENT(INOUT) :: info 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 12MPI_Info_get(info, key, valuelen, value, flag, ierror) 13 TYPE(MPI_Info), INTENT(IN) :: info 14CHARACTER(LEN=*), INTENT(IN) :: key 15INTEGER, INTENT(IN) :: valuelen 16CHARACTER(LEN=valuelen), INTENT(OUT) :: value 17LOGICAL, INTENT(OUT) :: flag 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20MPI_Info_get_nkeys(info, nkeys, ierror) 21TYPE(MPI_Info), INTENT(IN) :: info 22 INTEGER, INTENT(OUT) :: nkeys 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI_Info_get_nthkey(info, n, key, ierror) 25TYPE(MPI_Info), INTENT(IN) :: info 26INTEGER, INTENT(IN) :: n 27CHARACTER(LEN=*), INTENT(OUT) :: key 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) 31TYPE(MPI_Info), INTENT(IN) :: info 32 CHARACTER(LEN=*), INTENT(IN) :: key 33 INTEGER, INTENT(OUT) :: valuelen 34 LOGICAL, INTENT(OUT) :: flag 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI_Info_set(info, key, value, ierror) 37 TYPE(MPI_Info), INTENT(IN) :: info 38 CHARACTER(LEN=*), INTENT(IN) :: key, value 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 42A.4.8 Process Creation and Management Fortran 2008 Bindings 43 MPI_Close_port(port_name, ierror) 44 CHARACTER(LEN=*), INTENT(IN) :: port_name 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) 48

```
1
 CHARACTER(LEN=*), INTENT(IN) :: port_name
2
 TYPE(MPI_Info), INTENT(IN) :: info
3
 INTEGER, INTENT(IN) :: root
4
 TYPE(MPI_Comm), INTENT(IN) :: comm
5
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
8
 CHARACTER(LEN=*), INTENT(IN) :: port_name
9
 TYPE(MPI_Info), INTENT(IN) :: info
10
 INTEGER, INTENT(IN) :: root
11
 TYPE(MPI_Comm), INTENT(IN) :: comm
12
 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
 MPI_Comm_disconnect(comm, ierror)
16
 TYPE(MPI_Comm), INTENT(INOUT) :: comm
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
 MPI_Comm_get_parent(parent, ierror)
19
 TYPE(MPI_Comm), INTENT(OUT) :: parent
20
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
 MPI_Comm_join(fd, intercomm, ierror)
23
 INTEGER, INTENT(IN) :: fd
^{24}
 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
27
 array_of_errcodes, ierror)
28
 CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
29
 INTEGER, INTENT(IN) :: maxprocs, root
30
 TYPE(MPI_Info), INTENT(IN) :: info
31
 TYPE(MPI_Comm), INTENT(IN) :: comm
32
 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
33
 INTEGER :: array_of_errcodes(*)
34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
37
 array_of_maxprocs, array_of_info, root, comm, intercomm,
38
 array_of_errcodes, ierror)
39
 INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
40
 CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*)
41
 CHARACTER(LEN=*), INTENT(IN) :: array_of_argv(count, *)
42
 TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
43
 TYPE(MPI_Comm), INTENT(IN) :: comm
44
 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
45
 INTEGER :: array_of_errcodes(*)
46
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
 MPI_Lookup_name(service_name, info, port_name, ierror)
48
```

1 CHARACTER(LEN=*), INTENT(IN) :: service_name  $\mathbf{2}$ TYPE(MPI_Info), INTENT(IN) :: info 3 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 5MPI_Open_port(info, port_name, ierror) 6 TYPE(MPI_Info), INTENT(IN) :: info 7 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10MPI_Publish_name(service_name, info, port_name, ierror) 11 TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name 1213 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14MPI_Unpublish_name(service_name, info, port_name, ierror) 15CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name 16TYPE(MPI_Info), INTENT(IN) :: info 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19 20A.4.9 One-Sided Communications Fortran 2008 Bindings 21MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank, 22 target_disp, target_count, target_datatype, op, win, ierror) 23TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 24INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 25TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 26INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 27TYPE(MPI_Op), INTENT(IN) :: op 28 TYPE(MPI_Win), INTENT(IN) :: win 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, 32 target_rank, target_disp, win, ierror) 33 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 34 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: compare_addr 35 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 36 TYPE(MPI_Datatype), INTENT(IN) :: datatype 37 INTEGER, INTENT(IN) :: target_rank 38 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 39 TYPE(MPI_Win), INTENT(IN) :: win 40INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank, 42target_disp, op, win, ierror) 43 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 44TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 45TYPE(MPI_Datatype), INTENT(IN) :: datatype 46INTEGER, INTENT(IN) :: target_rank 47INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 48

```
1
 TYPE(MPI_Op), INTENT(IN) :: op
2
 TYPE(MPI_Win), INTENT(IN) :: win
3
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
4
 MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
5
 result_count, result_datatype, target_rank, target_disp,
6
 target_count, target_datatype, op, win, ierror)
7
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
8
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
9
 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
10
 target_count
11
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
12
 result_datatype
13
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
14
 TYPE(MPI_Op), INTENT(IN) :: op
15
 TYPE(MPI_Win), INTENT(IN) :: win
16
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
17
18
 MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
19
 target_disp, target_count, target_datatype, win, ierror)
20
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
21
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
22
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
23
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
24
 TYPE(MPI_Win), INTENT(IN) :: win
25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
 MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
27
 target_disp, target_count, target_datatype, win, ierror)
28
 TYPE(*), DIMENSION(..), INTENT(IN), 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
35
 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
36
 target_disp, target_count, target_datatype, op, win, request,
37
 ierror)
38
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
39
 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
40
 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
41
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
42
 TYPE(MPI_Op), INTENT(IN) :: op
43
 TYPE(MPI_Win), INTENT(IN) :: win
44
 TYPE(MPI_Request), INTENT(OUT) :: request
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
47
 result_addr, result_count, result_datatype, target_rank,
48
```

1 target_disp, target_count, target_datatype, op, win, request, 2 ierror) 3 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr 4 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr INTEGER, INTENT(IN) :: origin_count, result_count, target_rank, 56 target_count 7 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype, 8 result_datatype INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 9 10 TYPE(MPI_Op), INTENT(IN) :: op 11 TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Request), INTENT(OUT) :: request 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank, 15target_disp, target_count, target_datatype, win, request, 16 ierror) 17TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr 18 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 19 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 20INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 21TYPE(MPI_Win), INTENT(IN) :: win 22 TYPE(MPI_Request), INTENT(OUT) :: request 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2425MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank, 26target_disp, target_count, target_datatype, win, request, 27ierror) 28 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 29INTEGER, INTENT(IN) :: origin_count, target_rank, target_count TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 30 31INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 32 TYPE(MPI_Win), INTENT(IN) :: win 33 TYPE(MPI_Request), INTENT(OUT) :: request 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror) 36 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 37 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 38 INTEGER, INTENT(IN) :: disp_unit 39 TYPE(MPI_Info), INTENT(IN) :: info 40 TYPE(MPI_Comm), INTENT(IN) :: comm 41 TYPE(C_PTR), INTENT(OUT) :: baseptr 42TYPE(MPI_Win), INTENT(OUT) :: win 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4445MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) 46USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 47INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size 48

```
1
 INTEGER, INTENT(IN) :: disp_unit
2
 TYPE(MPI_Info), INTENT(IN) :: info
3
 TYPE(MPI_Comm), INTENT(IN) :: comm
4
 TYPE(C_PTR), INTENT(OUT) :: baseptr
5
 TYPE(MPI_Win), INTENT(OUT) :: win
6
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
 MPI_Win_attach(win, base, size, ierror)
8
 TYPE(MPI_Win), INTENT(IN) :: win
9
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
10
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
11
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
 MPI_Win_complete(win, ierror)
14
 TYPE(MPI_Win), INTENT(IN) :: win
15
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
16
 MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
17
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
18
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
19
 INTEGER, INTENT(IN) :: disp_unit
20
 TYPE(MPI_Info), INTENT(IN) :: info
21
 TYPE(MPI_Comm), INTENT(IN) :: comm
22
 TYPE(MPI_Win), INTENT(OUT) :: win
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
 MPI_Win_create_dynamic(info, comm, win, ierror)
26
 TYPE(MPI_Info), INTENT(IN) :: info
27
 TYPE(MPI_Comm), INTENT(IN) :: comm
28
 TYPE(MPI_Win), INTENT(OUT) :: win
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
 MPI_Win_detach(win, base, ierror)
^{31}
 TYPE(MPI_Win), INTENT(IN) :: win
32
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
 MPI_Win_fence(assert, win, ierror)
36
 INTEGER, INTENT(IN) :: assert
37
 TYPE(MPI_Win), INTENT(IN) :: win
38
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
39
 MPI_Win_flush_all(win, ierror)
40
 TYPE(MPI_Win), INTENT(IN) :: win
41
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
42
43
 MPI_Win_flush_local_all(win, ierror)
44
 TYPE(MPI_Win), INTENT(IN) :: win
45
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
 MPI_Win_flush_local(rank, win, ierror)
47
 INTEGER, INTENT(IN) :: rank
48
```

TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
<pre>MPI_Win_flush(rank, win, ierror)     INTEGER, INTENT(IN) :: rank     TYPE(MPI_Win), INTENT(IN) :: win     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	3 4 5 6 7
<pre>MPI_Win_free(win, ierror)     TYPE(MPI_Win), INTENT(INOUT) :: win     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	8 9 10 11
<pre>MPI_Win_get_group(win, group, ierror)     TYPE(MPI_Win), INTENT(IN) :: win     TYPE(MPI_Group), INTENT(OUT) :: group     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	11 12 13 14 15
<pre>MPI_Win_get_info(win, info_used, ierror)     TYPE(MPI_Win), INTENT(IN) :: win     TYPE(MPI_Info), INTENT(OUT) :: info_used     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	16 17 18 19 20
<pre>MPI_Win_lock_all(assert, win, ierror)     INTEGER, INTENT(IN) :: assert     TYPE(MPI_Win), INTENT(IN) :: win     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	21 22 23 24
<pre>MPI_Win_lock(lock_type, rank, assert, win, ierror)     INTEGER, INTENT(IN) :: lock_type, rank, assert     TYPE(MPI_Win), INTENT(IN) :: win     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	25 26 27 28
<pre>MPI_Win_post(group, assert, win, ierror)    TYPE(MPI_Group), INTENT(IN) :: group    INTEGER, INTENT(IN) :: assert    TYPE(MPI_Win), INTENT(IN) :: win    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	29 30 31 32 33 34
<pre>MPI_Win_set_info(win, info, ierror)     TYPE(MPI_Win), INTENT(IN) :: win     TYPE(MPI_Info), INTENT(IN) :: info     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	35 36 37 38 39
<pre>MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(MPI_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: rank INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size</pre>	40 41 42 43 44
INTEGER, INTENT(OUT) :: disp_unit TYPE(C_PTR), INTENT(OUT) :: baseptr INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45 46 47 48

```
1
 MPI_Win_start(group, assert, win, ierror)
\mathbf{2}
 TYPE(MPI_Group), INTENT(IN) :: group
3
 INTEGER, INTENT(IN) :: assert
4
 TYPE(MPI_Win), INTENT(IN) :: win
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_Win_sync(win, ierror)
7
 TYPE(MPI_Win), INTENT(IN) :: win
8
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
9
10
 MPI_Win_test(win, flag, ierror)
11
 TYPE(MPI_Win), INTENT(IN) :: win
12
 LOGICAL, INTENT(OUT) :: flag
13
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
 MPI_Win_unlock_all(win, ierror)
15
 TYPE(MPI_Win), INTENT(IN) :: win
16
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
17
18
 MPI_Win_unlock(rank, win, ierror)
19
 INTEGER, INTENT(IN) :: rank
20
 TYPE(MPI_Win), INTENT(IN) :: win
21
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
22
 MPI_Win_wait(win, ierror)
23
 TYPE(MPI_Win), INTENT(IN) :: win
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
27
 A.4.10 External Interfaces Fortran 2008 Bindings
28
 MPI_Grequest_complete(request, ierror)
29
 TYPE(MPI_Request), INTENT(IN) :: request
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
 MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
33
 ierror)
34
 PROCEDURE(MPI_Grequest_query_function) :: query_fn
35
 PROCEDURE(MPI_Grequest_free_function) :: free_fn
36
 PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
37
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
38
 TYPE(MPI_Request), INTENT(OUT) :: request
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_Init_thread(required, provided, ierror)
42
 INTEGER, INTENT(IN) :: required
 INTEGER, INTENT(OUT) :: provided
43
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
 MPI_Is_thread_main(flag, ierror)
46
 LOGICAL, INTENT(OUT) :: flag
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_Query_thread(provided, ierror) INTEGER, INTENT(OUT) :: provided	1 2 3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Status_set_cancelled(status, flag, ierror)	5
TYPE(MPI_Status), INTENT(INOUT) :: status	6
LOGICAL, INTENT(IN) :: flag	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
MPI_Status_set_elements(status, datatype, count, ierror)	9
TYPE(MPI_Status), INTENT(INOUT) :: status	10
TYPE(MPI_Datatype), INTENT(IN) :: datatype	11
INTEGER, INTENT(IN) :: count	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Status_set_elements_x(status, datatype, count, ierror)	14
TYPE(MPI_Status), INTENT(INOUT) :: status	15
TYPE(MPI_Datatype), INTENT(IN) :: datatype	16
INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) :: count	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18 19
	20
	20
A.4.11 I/O Fortran 2008 Bindings	22
MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,	23
extra_state, ierror)	24
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	25
TYPE(C_PTR), VALUE :: userbuf, filebuf	26
TYPE(MPI_Datatype) :: datatype	27
INTEGER :: count, ierror	28
INTEGER(KIND=MPI_OFFSET_KIND) :: position	29
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	30
MPI_File_close(fh, ierror)	31
TYPE(MPI_File), INTENT(INOUT) :: fh	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33 34
MPI_File_delete(filename, info, ierror)	35
CHARACTER(LEN=*), INTENT(IN) :: filename	36
TYPE(MPI_Info), INTENT(IN) :: info	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
	39
MPI_File_get_amode(fh, amode, ierror)	40
TYPE(MPI_File), INTENT(IN) :: fh	41
INTEGER, INTENT(OUT) :: amode INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
INIEGER, OFIIONAE, INIENI(001/ IEIIOI	43
MPI_File_get_atomicity(fh, flag, ierror)	44
TYPE(MPI_File), INTENT(IN) :: fh	45
LOGICAL, INTENT(OUT) :: flag	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
	48

```
1
 MPI_File_get_byte_offset(fh, offset, disp, ierror)
\mathbf{2}
 TYPE(MPI_File), INTENT(IN) :: fh
3
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
4
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
5
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
 MPI_File_get_group(fh, group, ierror)
7
 TYPE(MPI_File), INTENT(IN) :: fh
8
 TYPE(MPI_Group), INTENT(OUT) :: group
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_File_get_info(fh, info_used, ierror)
12
 TYPE(MPI_File), INTENT(IN) :: fh
13
 TYPE(MPI_Info), INTENT(OUT) :: info_used
14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
 MPI_File_get_position(fh, offset, ierror)
16
 TYPE(MPI_File), INTENT(IN) :: fh
17
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
18
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
 MPI_File_get_position_shared(fh, offset, ierror)
21
 TYPE(MPI_File), INTENT(IN) :: fh
22
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_File_get_size(fh, size, ierror)
25
 TYPE(MPI_File), INTENT(IN) :: fh
26
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
27
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
 MPI_File_get_type_extent(fh, datatype, extent, ierror)
30
 TYPE(MPI_File), INTENT(IN) :: fh
31
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
 extent
33
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
 MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
35
 TYPE(MPI_File), INTENT(IN) :: fh
36
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::
 disp
37
 TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
38
 CHARACTER(LEN=*), INTENT(OUT) :: datarep
39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
 MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
42
 TYPE(MPI_File), INTENT(IN) :: fh
43
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
44
 INTEGER, INTENT(IN) :: count
45
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
 TYPE(MPI_Request), INTENT(OUT) ::
 request
47
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
48
```

1 MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror) 2 TYPE(MPI_File), INTENT(IN) :: fh 3 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 4 INTEGER, INTENT(IN) :: count 56 TYPE(MPI_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI_Request), INTENT(OUT) :: request 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) 10 TYPE(MPI_File), INTENT(IN) :: fh 11 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 12TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 13 INTEGER, INTENT(IN) :: count 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15TYPE(MPI_Request), INTENT(OUT) :: request 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 MPI_File_iread(fh, buf, count, datatype, request, ierror) 19 TYPE(MPI_File), INTENT(IN) :: fh 20TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 21INTEGER, INTENT(IN) :: count 22 TYPE(MPI_Datatype), INTENT(IN) :: datatype 23TYPE(MPI_Request), INTENT(OUT) :: request 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) 26TYPE(MPI_File), INTENT(IN) :: fh 27TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 28 INTEGER, INTENT(IN) :: count 29 TYPE(MPI_Datatype), INTENT(IN) :: datatype 30 TYPE(MPI_Request), INTENT(OUT) :: request 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror) 34 TYPE(MPI_File), INTENT(IN) :: fh 35 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 36 INTEGER, INTENT(IN) :: count 37 TYPE(MPI_Datatype), INTENT(IN) :: datatype 38 TYPE(MPI_Request), INTENT(OUT) :: request 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror) 41 TYPE(MPI_File), INTENT(IN) :: fh 42INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 43 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 44INTEGER, INTENT(IN) :: count 45TYPE(MPI_Datatype), INTENT(IN) :: datatype 46TYPE(MPI_Request), INTENT(OUT) :: request 47INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

```
1
 MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
\mathbf{2}
 TYPE(MPI_File), INTENT(IN) :: fh
3
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
4
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
5
 INTEGER, INTENT(IN) :: count
6
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
 TYPE(MPI_Request), INTENT(OUT) :: request
8
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
 MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
10
 TYPE(MPI_File), INTENT(IN) :: fh
11
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
12
 INTEGER, INTENT(IN) :: count
13
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
 TYPE(MPI_Request), INTENT(OUT) ::
 request
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
20
 INTEGER, INTENT(IN) :: count
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 TYPE(MPI_Request), INTENT(OUT) ::
 request
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_File_open(comm, filename, amode, info, fh, ierror)
25
 TYPE(MPI_Comm), INTENT(IN) :: comm
26
 CHARACTER(LEN=*), INTENT(IN) :: filename
27
 INTEGER, INTENT(IN) :: amode
28
 TYPE(MPI_Info), INTENT(IN) :: info
29
 TYPE(MPI_File), INTENT(OUT) :: fh
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
 MPI_File_preallocate(fh, size, ierror)
33
 TYPE(MPI_File), INTENT(IN) :: fh
34
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
 MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
37
 TYPE(MPI_File), INTENT(IN) :: fh
38
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
39
 INTEGER, INTENT(IN) :: count
40
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
 INTEGER, OPTIONAL, INTENT(OUT) ::
 ierror
42
43
 MPI_File_read_all_end(fh, buf, status, ierror)
44
 TYPE(MPI_File), INTENT(IN) :: fh
45
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
46
 TYPE(MPI_Status) :: status
47
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_File_read_all(fh, buf, count, datatype, status, ierror)
 2
 TYPE(MPI_File), INTENT(IN) :: fh
 3
 TYPE(*), DIMENSION(..) :: buf
 INTEGER, INTENT(IN) :: count
 4
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 5
 6
 TYPE(MPI_Status) :: status
 7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 8
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
 9
 TYPE(MPI_File), INTENT(IN) :: fh
 10
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 11
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 12
 INTEGER, INTENT(IN) :: count
 13
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 14
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16
MPI_File_read_at_all_end(fh, buf, status, ierror)
 17
 TYPE(MPI_File), INTENT(IN) :: fh
 18
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 19
 TYPE(MPI_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 20
 21
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
 22
 TYPE(MPI_File), INTENT(IN) :: fh
 23
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 24
 TYPE(*), DIMENSION(..) :: buf
 25
 INTEGER, INTENT(IN) :: count
 26
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 27
 TYPE(MPI_Status) :: status
 28
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 29
 30
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
 31
 TYPE(MPI_File), INTENT(IN) :: fh
 32
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
 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
MPI_File_read(fh, buf, count, datatype, status, ierror)
 39
 TYPE(MPI_File), INTENT(IN) :: fh
 40
 TYPE(*), DIMENSION(..) :: buf
 41
 INTEGER, INTENT(IN) :: count
 42
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 43
 TYPE(MPI_Status) :: status
 44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 45
 46
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
 47
 TYPE(MPI_File), INTENT(IN) :: fh
 48
```

```
1
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
2
 INTEGER, INTENT(IN) :: count
3
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
 MPI_File_read_ordered_end(fh, buf, status, ierror)
6
 TYPE(MPI_File), INTENT(IN) :: fh
7
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
8
 TYPE(MPI_Status) :: status
9
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
 MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
12
 TYPE(MPI_File), INTENT(IN) :: fh
13
 TYPE(*), DIMENSION(..) :: buf
14
 INTEGER, INTENT(IN) :: count
15
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
 TYPE(MPI_Status) :: status
17
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
 MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
19
 TYPE(MPI_File), INTENT(IN) :: fh
20
 TYPE(*), DIMENSION(..) :: buf
21
 INTEGER, INTENT(IN) :: count
22
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
 TYPE(MPI_Status) :: status
24
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
 MPI_File_seek(fh, offset, whence, ierror)
27
 TYPE(MPI_File), INTENT(IN) :: fh
28
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29
 INTEGER, INTENT(IN) :: whence
30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
 MPI_File_seek_shared(fh, offset, whence, ierror)
32
 TYPE(MPI_File), INTENT(IN) :: fh
33
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
34
 INTEGER, INTENT(IN) :: whence
35
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
 MPI_File_set_atomicity(fh, flag, ierror)
38
 TYPE(MPI_File), INTENT(IN) :: fh
39
 LOGICAL, INTENT(IN) :: flag
40
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
 MPI_File_set_info(fh, info, ierror)
42
 TYPE(MPI_File), INTENT(IN) :: fh
43
 TYPE(MPI_Info), INTENT(IN) :: info
44
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
 MPI_File_set_size(fh, size, ierror)
47
 TYPE(MPI_File), INTENT(IN) :: fh
48
```

INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2
	3
<pre>MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)</pre>	4
TYPE(MPI_File), INTENT(IN) :: fh	5
<pre>INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp</pre>	6
TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype	7
CHARACTER(LEN=*), INTENT(IN) :: datarep	8
TYPE(MPI_Info), INTENT(IN) :: info	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_File_sync(fh, ierror)	12
TYPE(MPI_File), INTENT(IN) :: fh	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)	15
TYPE(MPI_File), INTENT(IN) :: fh	16
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	10
INTEGER, INTENT(IN) :: count	18
TYPE(MPI_Datatype), INTENT(IN) :: datatype	19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	20
MPI_File_write_all_end(fh, buf, status, ierror)	21
TYPE(MPI_File), INTENT(IN) :: fh	22
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	23 24
TYPE(MPI_Status) :: status	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25 26
MPI_File_write_all(fh, buf, count, datatype, status, ierror)	20
TYPE(MPI_File), INTENT(IN) :: fh	28
TYPE(*), DIMENSION(), INTENT(IN) :: buf	28 29
INTEGER, INTENT(IN) :: count	30
TYPE(MPI_Datatype), INTENT(IN) :: datatype	31
TYPE(MPI_Status) :: status	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
	34
<pre>MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)</pre>	35
TYPE(MPI_File), INTENT(IN) :: fh	36
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	30
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	38
INTEGER, INTENT(IN) :: count	39
TYPE(MPI_Datatype), INTENT(IN) :: datatype	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
MPI_File_write_at_all_end(fh, buf, status, ierror)	41 42
TYPE(MPI_File), INTENT(IN) :: fh	
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	43
TYPE(MPI_Status) :: status	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
INIEGEN, OFFICENE, INTENT(UOT) ICITOL	46
<pre>MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)</pre>	47
	48

```
1
 TYPE(MPI_File), INTENT(IN) :: fh
2
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
3
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
4
 INTEGER, INTENT(IN) :: count
5
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
 TYPE(MPI_Status) :: status
7
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
 MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
9
 TYPE(MPI_File), INTENT(IN) :: fh
10
 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
11
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
12
 INTEGER, INTENT(IN) :: count
13
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
 TYPE(MPI_Status) :: status
15
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
 MPI_File_write(fh, buf, count, datatype, status, ierror)
18
 TYPE(MPI_File), INTENT(IN) :: fh
19
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
20
 INTEGER, INTENT(IN) :: count
21
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
 TYPE(MPI_Status) :: status
23
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
 MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
25
 TYPE(MPI_File), INTENT(IN) :: fh
26
 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
27
 INTEGER, INTENT(IN) :: count
28
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
 MPI_File_write_ordered_end(fh, buf, status, ierror)
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
 MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
37
 TYPE(MPI_File), INTENT(IN) :: fh
38
 TYPE(*), DIMENSION(...), INTENT(IN) :: buf
39
 INTEGER, INTENT(IN) :: count
40
 TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
 TYPE(MPI_Status) :: status
42
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
^{44}
 MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
45
 TYPE(MPI_File), INTENT(IN) :: fh
46
 TYPE(*), DIMENSION(...), INTENT(IN) :: 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_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
 5
 dtype_file_extent_fn, extra_state, ierror)
 6
 CHARACTER(LEN=*), INTENT(IN) :: datarep
 7
 PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
 8
 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
 14
A.4.12 Language Bindings Fortran 2008 Bindings
 15
MPI_F_sync_reg(buf)
 16
 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
 17
 18
MPI_Sizeof(x, size, ierror)
 19
 TYPE(*), DIMENSION(..) :: x
 20
 INTEGER, INTENT(OUT) :: size
 21
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 22
MPI_Status_f082f(f08_status, f_status, ierror)
 23
 TYPE(MPI_Status), INTENT(IN) :: f08_status
 24
 INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
 25
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
 27
MPI_Status_f2f08(f_status, f08_status, ierror)
 28
 INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
 29
 TYPE(MPI_Status), INTENT(OUT) :: f08_status
 30
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 31
MPI_Type_create_f90_complex(p, r, newtype, ierror)
 32
 INTEGER, INTENT(IN) :: p, r
 33
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 34
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 35
 36
MPI_Type_create_f90_integer(r, newtype, ierror)
 37
 INTEGER, INTENT(IN) :: r
 38
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 39
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 40
 41
MPI_Type_create_f90_real(p, r, newtype, ierror)
 42
 INTEGER, INTENT(IN) :: p, r
 TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 43
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 44
 45
MPI_Type_match_size(typeclass, size, datatype, ierror)
 46
 INTEGER, INTENT(IN) :: typeclass, size
 47
 TYPE(MPI_Datatype), INTENT(OUT) :: datatype
 48
```

```
1
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
3
 A.4.13 Tools / Profiling Interface Fortran 2008 Bindings
4
\mathbf{5}
 MPI_Pcontrol(level)
6
 INTEGER, INTENT(IN) :: level
\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
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	801
A.5 Fortran Bindings with mpif.h or the mpi Module	1
A.5.1 Point-to-Point Communication Fortran Bindings	2
, and the second s	4
<pre>MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)</pre>	5
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	6
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	7
<pre><type> BUF(*)</type></pre>	9
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	10
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)	11
<pre><type> BUFFER(*)</type></pre>	12 13
INTEGER SIZE, IERROR	13
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)	15
<type> BUFFER_ADDR(*)</type>	16
INTEGER SIZE, IERROR	17
MPI_CANCEL(REQUEST, IERROR)	18 19
INTEGER REQUEST, IERROR	20
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)	21
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	22
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	23
<type> BUF(*)</type>	24 25
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	26
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)	27
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	
LOGICAL FLAG	29 30
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)	30
<type> BUF(*)</type>	32
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR	33
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)	34
LOGICAL FLAG	35 36
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	37
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	38
<pre><type> BUF(*) INTEGED COUNT DATATYDE COUDCE TAC COMM DECUEST LEDDOD</type></pre>	39
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	40 41
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	41 42
<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>	43
	44
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	45
<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>	46 47
· · · · · · · · · · · · · · · · · · ·	48

```
1
 MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
\mathbf{2}
 <type> BUF(*)
3
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4
 MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
5
 INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
6
7
 MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
8
 <type> BUF(*)
9
 INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
10
 MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
11
 INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
12
13
 MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
14
 <type> BUF(*)
15
 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
16
 IERROR
17
 MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
18
 <type> BUF(*)
19
 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
20
21
 MPI_REQUEST_FREE(REQUEST, IERROR)
22
 INTEGER REQUEST, IERROR
23
 MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
24
 INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
25
 LOGICAL FLAG
26
27
 MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
28
 <type> BUF(*)
29
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
30
 MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
^{31}
 <type> BUF(*)
32
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
33
34
 MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
35
 <type> BUF(*)
36
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
37
 MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
38
 <type> BUF(*)
39
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
40
41
 MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
42
 COMM, STATUS, IERROR)
43
 <type> BUF(*)
44
 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
45
 STATUS(MPI_STATUS_SIZE), IERROR
46
47
 MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
48
```

<type> SENDBUF(*), RECVBUF(*)</type>	1
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,	2 3
SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	4
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	5
<type> BUF(*)</type>	6
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	7
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	8
<type> BUF(*)</type>	9
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	10
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)	11 12
INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR	13
MPI_START(REQUEST, IERROR)	14
INTEGER REQUEST, IERROR	15
	16
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)	17
LOGICAL FLAG INTEGER COUNT, ARRAY_OF_REQUESTS(*),	18
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR	19
	20 21
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)	22
LOGICAL FLAG INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),	23
INTEGER COUNT, ARRAT_OF_REQUESTS(*), INDEX, STATUS(MFT_STATUS_SIZE), IERROR	24
	25
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR) LOGICAL FLAG	26
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	27
	28 29
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)	30
LOGICAL FLAG	31
INTEGER 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(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR	35
ARRAY_UF_STATUSES(MP1_STATUS_STZE,*), TERRUR	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	40
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)	41
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),	42
IERROR	43
MPI_WAIT(REQUEST, STATUS, IERROR)	44
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	45 46
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,	46 47
ARRAY_OF_STATUSES, IERROR)	48

```
1
 INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
2
 ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
3
4
 A.5.2 Datatypes Fortran Bindings
5
6
 INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
7
 INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
8
 INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)
9
 INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2
10
11
 MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
12
 <type> LOCATION(*)
13
 INTEGER IERROR
14
 INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
15
 MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
16
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
17
18
 MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
19
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
20
 INTEGER(KIND=MPI_COUNT_KIND) COUNT
21
 MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
22
 POSITION, IERROR)
23
 INTEGER INCOUNT, DATATYPE, IERROR
24
 INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
25
26
 CHARACTER*(*) DATAREP
 <type> INBUF(*), OUTBUF(*)
27
28
 MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
29
 INTEGER INCOUNT, DATATYPE, IERROR
30
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
31
 CHARACTER*(*) DATAREP
32
 MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
33
34
 <type> INBUF(*), OUTBUF(*)
 INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
35
36
 MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
37
 INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
38
39
 MPI_TYPE_COMMIT(DATATYPE, IERROR)
40
 INTEGER DATATYPE, IERROR
41
 MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
42
 INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
43
^{44}
 MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
45
 ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
46
 OLDTYPE, NEWTYPE, IERROR)
47
 INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
48
 ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,
```

NEWTYPE, IERROR

	0
MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	2 3
	4
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	5
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	6
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,	7
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	8
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR	9
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	10
	11
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,	12
IERROR)	13
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR	14
INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE	15
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	16
OLDTYPE, NEWTYPE, IERROR)	17
INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,	18
NEWTYPE, IERROR	19
	20
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)	21
INTEGER OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	22
INTEGER(KIND-FRII_RDDRESS_KIND) ED; EKIENI	23
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,	24
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)	25 26
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,	20 27
IERROR	21
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	29
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,	30
ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)	31
<pre>INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),</pre>	32
ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR	33
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)	34
INTEGER OLDTYPE, NEWTYPE, IERROR	35
INTEGER OLDTITE, NEWITTE, IERROR	36
MPI_TYPE_FREE(DATATYPE, IERROR)	37
INTEGER DATATYPE, IERROR	38
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	39
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	40
IERROR)	41
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	42
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	43
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)	44
	45
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,	46
COMBINER, IERROR)	47
INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,	48

1

1	IERROR
2 3	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
4	INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
5 6	MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
7	INTEGER DATATYPE, IERROR
8 9	INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
10	MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
11	INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
12 13	MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
14	INTEGER DATATYPE, IERROR
15 16	INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
17	MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
18	OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
19 20	OLDTYPE, NEWTYPE, IERROR
21	MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
22 23	INTEGER DATATYPE, SIZE, IERROR
24	MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
25 26	INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) SIZE
20 27	MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
28 29	INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
30 31	MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, IERROR)
32	INTEGER OUTCOUNT, DATATYPE, IERROR
33	INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
$\frac{34}{35}$	CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*)</type>
36	MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
37	IERROR)
38 39	<type> INBUF(*), OUTBUF(*)</type>
40	INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
41 42	A.F.2. Collective Communication Fortran Dindings
43	A.5.3 Collective Communication Fortran Bindings
44	MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
45 46	<type> SENDBUF(*), RECVBUF(*)</type>
47	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
48	IERROR

MFI_ALLGAINER(SEMDBOF, SEMDCOONI, SEMDTIFE, ALCODOF, ALCOCOONI, ALCOTIFE,	1 2
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	3 4
MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	5 6 7
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	8 9 10
RECVTYPE, COMM, IERROR)	11 12 13
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, IERROR	14 15
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,	16 17 18
<pre><type> SENDBUF(*), RECVBUF(*) INTEGEP COUNT DATATVDE OD COMM INFO PEOUEST IEPPOP</type></pre>	19 20
<pre> type&gt; SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR </pre>	21 22 23
<pre>MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR</type></pre>	24 25 26 27 28 29
<pre>MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type></pre>	30 31 32 33
<pre>MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), DEGUTYPE COMM INTO DEGUTET IERDOD</type></pre>	34 35 36 37 38 39
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, IERROR</type>	40 41 42 43 44
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)	45 46 47 48

```
1
 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
2
 RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
3
 MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
4
 RDISPLS, RECVTYPES, COMM, IERROR)
5
 <type> SENDBUF(*), RECVBUF(*)
6
 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
7
 RDISPLS(*), RECVTYPES(*), COMM, IERROR
8
9
 MPI_BARRIER(COMM, IERROR)
10
 INTEGER COMM, IERROR
11
 MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
12
 INTEGER COMM, INFO, REQUEST, IERROR
13
14
 MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
15
 <type> BUFFER(*)
16
 INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
17
 MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
18
 <type> BUFFER(*)
19
 INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
20
21
 MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
22
 IERROR)
23
 <type> SENDBUF(*), RECVBUF(*)
^{24}
 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
25
 MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
26
 <type> SENDBUF(*), RECVBUF(*)
27
 INTEGER COUNT, DATATYPE, OP, COMM, IERROR
28
29
 MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
30
 ROOT, COMM, INFO, REQUEST, IERROR)
31
 <type> SENDBUF(*), RECVBUF(*)
32
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
33
 REQUEST, IERROR
34
 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
35
 ROOT, COMM, IERROR)
36
 <type> SENDBUF(*), RECVBUF(*)
37
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
38
39
 MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
40
 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
41
 <type> SENDBUF(*), RECVBUF(*)
42
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
43
 COMM, INFO, REQUEST, IERROR
44
 MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
45
 RECVTYPE, ROOT, COMM, IERROR)
46
 <type> SENDBUF(*), RECVBUF(*)
47
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
48
```

COMM, IERROR

	2
MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>	2 3 4 5
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	6
<pre>MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</type></pre>	7 8 9 10 11 12
MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type>	12 13 14 15 16
MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type>	17 18 19 20
<pre>MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</type></pre>	21 22 23 24 25 26
<pre>MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>	27 28 29 30 31 32
MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR	33 34
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR</type>	35 36 37 38
MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type>	39 40 41
<pre>MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR</type></pre>	42 43 44 45 46
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	47 48

Unofficial Draft for Comment Only

1

1RECVTYPE, ROOT, COMM, REQUEST, IERROR)  $\mathbf{2}$ <type> SENDBUF(*), RECVBUF(*) 3 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, 4 COMM, REQUEST, IERROR 5MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 6 REQUEST, IERROR) 7 <type> SENDBUF(*), RECVBUF(*) 8 INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR 9 10MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 11 REQUEST, IERROR) 12<type> SENDBUF(*), RECVBUF(*) 13INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR 14MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, 15IERROR) 16<type> SENDBUF(*), RECVBUF(*) 17INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR 18 19 MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 20<type> SENDBUF(*), RECVBUF(*) 21INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 22 MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 23ROOT, COMM, REQUEST, IERROR) 24<type> SENDBUF(*), RECVBUF(*) 25INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 26IERROR 2728MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 29RECVTYPE, ROOT, COMM, REQUEST, IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 32 COMM, REQUEST, IERROR 33 MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR) 34 LOGICAL COMMUTE 35 INTEGER OP, IERROR 36 37 MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR) 38 EXTERNAL USER_FN 39 LOGICAL COMMUTE 40INTEGER OP, IERROR 41 MPI_OP_FREE(OP, IERROR) 42INTEGER OP, IERROR 43 44MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, 45REQUEST, IERROR) 46<type> SENDBUF(*), RECVBUF(*) 47 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 811	
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) <type> INBUF(*), INOUTBUF(*) INTEGER COUNT, DATATYPE, OP, IERROR</type>	1 2 3 4
<pre>MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR</type></pre>	4 5 6 7 8
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR</type>	9 10 11 12
MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR</type>	13 14 15 16 17
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR</type>	18 19 20 21
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR</type>	22 23 24 25
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR</type>	26 27 28 29
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR</type>	30 31 32 33
<pre>MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR</type></pre>	34 35 36 37 38 39
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR</type>	40 41 42 43
<pre>MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,</pre>	44 45 46 47 48

```
1
 COMM, INFO, REQUEST, IERROR
\mathbf{2}
 MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
3
 RECVTYPE, ROOT, COMM, IERROR)
4
 <type> SENDBUF(*), RECVBUF(*)
5
 INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
6
 COMM. IERROR
7
8
9
 A.5.4 Groups, Contexts, Communicators, and Caching Fortran Bindings
10
 MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
11
 INTEGER COMM1, COMM2, RESULT, IERROR
12
13
 MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
14
 INTEGER COMM, GROUP, NEWCOMM, IERROR
15
 MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
16
 INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
17
18
 MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
19
 EXTRA_STATE, IERROR)
20
 EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
21
 INTEGER COMM_KEYVAL, IERROR
22
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
23
 MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
^{24}
 INTEGER COMM, COMM_KEYVAL, IERROR
25
26
 MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
27
 INTEGER COMM, NEWCOMM, IERROR
28
 MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
29
30
 ATTRIBUTE_VAL_OUT, FLAG, IERROR)
 INTEGER OLDCOMM, COMM_KEYVAL, IERROR
31
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
32
33
 ATTRIBUTE_VAL_OUT
34
 LOGICAL FLAG
35
 MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
36
 INTEGER COMM, INFO, NEWCOMM, IERROR
37
38
 MPI_COMM_FREE(COMM, IERROR)
39
 INTEGER COMM, IERROR
40
 MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
41
 INTEGER COMM_KEYVAL, IERROR
42
43
 MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
44
 INTEGER COMM, COMM_KEYVAL, IERROR
45
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
46
 LOGICAL FLAG
47
 MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
48
```

INTEGER COMM, INFO_USED, IERROR	1
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)	2 3
INTEGER COMM, RESULTLEN, IERROR CHARACTER*(*) COMM_NAME	4
	5 6
MPI_COMM_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR	7
	8
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR	9 10
	10
MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR) INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR	12 13
MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	14
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	15
INTEGER OLDCOMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	16
ATTRIBUTE_VAL_OUT	17 18
LOGICAL FLAG	19
MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,	20
IERROR)	21
INTEGER COMM, COMM_KEYVAL, IERROR	22 23
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	24
MPI_COMM_RANK(COMM, RANK, IERROR)	25
INTEGER COMM, RANK, IERROR	26
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)	27 28
INTEGER COMM, GROUP, IERROR	29
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)	30
INTEGER COMM, SIZE, IERROR	31 32
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)	33
INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	34
	35
MPI_COMM_SET_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR	36 37
	38
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR) INTEGER COMM, IERROR	39
CHARACTER*(*) COMM_NAME	40
MPI_COMM_SIZE(COMM, SIZE, IERROR)	41 42
INTEGER COMM, SIZE, IERROR	43
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)	44
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	45
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	46 47
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR	48

1 2 3	MPI_COMM_TEST_INTER(COMM, FLAG, IERROR) INTEGER COMM, IERROR LOGICAL FLAG
4 5 6	MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR) INTEGER GROUP1, GROUP2, RESULT, IERROR
7 8	MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
9 10 11	MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
12 13	MPI_GROUP_FREE(GROUP, IERROR) INTEGER GROUP, IERROR
14 15 16	MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
17 18 19	MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
20 21	MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
22 23 24	MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
25 26	MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR
27 28 29	MPI_GROUP_SIZE(GROUP, SIZE, IERROR) INTEGER GROUP, SIZE, IERROR
30 31	MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
32 33 34	MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
35 36 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
39 40 41 42	MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH
43 44 45 46	MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL, EXTRA_STATE, IERROR) EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN INTEGER TYPE_KEYVAL, IERROR
47 48	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	815
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR	1 2
MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG	3 4 5 7 8 9
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR) INTEGER TYPE_KEYVAL, IERROR	10 11
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	12 13 14 15 16
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR CHARACTER*(*) TYPE_NAME	17 18 19 20
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_I ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG	
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STAT IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	28 29 30
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	31 32 33 34
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR) INTEGER DATATYPE, IERROR CHARACTER*(*) TYPE_NAME	35 36 37 38
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	39 40 41 42 43
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR	44 45 46
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	47 48

```
1
 INTEGER OLDWIN, WIN_KEYVAL, IERROR
\mathbf{2}
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
3
 ATTRIBUTE_VAL_OUT
4
 LOGICAL FLAG
5
 MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
6
 INTEGER WIN_KEYVAL, IERROR
7
8
 MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
9
 INTEGER WIN, WIN_KEYVAL, IERROR
10
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
11
 LOGICAL FLAG
12
 MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
13
 INTEGER WIN, RESULTLEN, IERROR
14
 CHARACTER*(*) WIN_NAME
15
16
 MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
17
 ATTRIBUTE_VAL_OUT, FLAG, IERROR)
18
 INTEGER OLDWIN, WIN_KEYVAL, IERROR
19
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
20
 ATTRIBUTE_VAL_OUT
21
 LOGICAL FLAG
22
 MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
23
 INTEGER WIN, WIN_KEYVAL, IERROR
24
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
25
26
 MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
27
 INTEGER WIN, WIN_KEYVAL, IERROR
28
 INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
29
 MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
30
 INTEGER WIN, IERROR
^{31}
 CHARACTER*(*) WIN_NAME
32
33
34
 A.5.5 Process Topologies Fortran Bindings
35
 MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
36
 INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
37
38
 MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
39
 INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
40
 LOGICAL PERIODS(*), REORDER
41
42
 MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
43
 INTEGER COMM, NDIMS, IERROR
44
 MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
45
 INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
46
 LOGICAL PERIODS(*)
47
48
 MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
```

INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR LOGICAL PERIODS(*)	1 $2$
MPI_CART_RANK(COMM, COORDS, RANK, IERROR) INTEGER COMM, COORDS(*), RANK, IERROR	3 4 5
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR) INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR	6 7
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR) INTEGER COMM, NEWCOMM, IERROR LOGICAL REMAIN_DIMS(*)	8 9 10 11
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) INTEGER NNODES, NDIMS, DIMS(*), IERROR	12 13
<pre>MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, COMM_DIST_GRAPH, IERROR) INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER</pre>	14 15 16 17 18 19 20 21
<pre>MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,</pre>	22 23 24 25 26
<pre>MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR</pre>	27 28 29 30 31
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) INTEGER COMM, INDEGREE, OUTDEGREE, IERROR LOGICAL WEIGHTED	32 33 34
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERROR) INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR LOGICAL REORDER	35 36 37 38 39
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR) INTEGER COMM, NNODES, NEDGES, IERROR	40 41 42
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	43 44
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR	45 46 47
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)	48

```
1
 INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
\mathbf{2}
 MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
3
 INTEGER COMM, RANK, NNEIGHBORS, IERROR
4
5
 MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
6
 RECVTYPE, COMM, REQUEST, IERROR)
7
 <type> SENDBUF(*), RECVBUF(*)
8
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
9
 MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
10
 DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
11
 <type> SENDBUF(*), RECVBUF(*)
12
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
13
 REQUEST, IERROR
14
15
 MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
16
 RECVTYPE, COMM, REQUEST, IERROR)
17
 <type> SENDBUF(*), RECVBUF(*)
18
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
19
 MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
20
 RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
21
 <type> SENDBUF(*), RECVBUF(*)
22
 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
23
 RECVTYPE, COMM, REQUEST, IERROR
^{24}
25
 MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
26
 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
27
 <type> SENDBUF(*), RECVBUF(*)
28
 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
29
 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
30
 REQUEST, IERROR
31
 MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
32
 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
33
 <type> SENDBUF(*), RECVBUF(*)
34
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
35
 IERROR
36
37
 MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
38
 RECVTYPE, COMM, IERROR)
39
 <type> SENDBUF(*), RECVBUF(*)
40
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
41
 MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
42
 RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
43
 <type> SENDBUF(*), RECVBUF(*)
44
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
45
 INFO, REQUEST, IERROR
46
47
 MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
48
```

1 DISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) 2 3 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, IERROR 5MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 6 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 7 <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 9 IERROR 1011 MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) 1213 <type> SENDBUF(*), RECVBUF(*) 14INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 15MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, 16RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, 17IERROR) 18 <type> SENDBUF(*), RECVBUF(*) 19 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 20RECVTYPE, COMM, INFO, REQUEST, IERROR 2122 MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 23RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) 24<type> SENDBUF(*), RECVBUF(*) 25INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 26RECVTYPE, COMM, IERROR 27MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, 28 RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, 29 IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 32 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 33 INFO, REQUEST, IERROR 34 35MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 36 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR) 37 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 39 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 40IERROR 41 MPI_TOPO_TEST(COMM, STATUS, IERROR) 42INTEGER COMM, STATUS, IERROR 43 4445A.5.6 MPI Environmental Management Fortran Bindings 46DOUBLE PRECISION MPI_WTICK() 4748

```
1
 DOUBLE PRECISION MPI_WTIME()
\mathbf{2}
 MPI_ABORT(COMM, ERRORCODE, IERROR)
3
 INTEGER COMM, ERRORCODE, IERROR
4
5
 MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
6
 INTEGER ERRORCLASS, IERROR
7
 MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
8
 INTEGER ERRORCLASS, ERRORCODE, IERROR
9
10
 MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
11
 INTEGER ERRORCODE, IERROR
12
 CHARACTER*(*) STRING
13
 MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
14
 INTEGER INFO, IERROR
15
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
16
17
 If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
18
 INTERFACE MPI_ALLOC_MEM
19
 SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
20
 IMPORT :: MPI_ADDRESS_KIND
21
 INTEGER :: INFO, IERROR
22
 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
23
 END SUBROUTINE
24
 SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
25
 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
26
 IMPORT :: MPI_ADDRESS_KIND
27
 INTEGER :: INFO, IERROR
28
 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
29
 TYPE(C_PTR) :: BASEPTR
30
 END SUBROUTINE
31
 END INTERFACE
32
 MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
33
 INTEGER COMM, ERRORCODE, IERROR
34
35
 MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
36
 EXTERNAL COMM_ERRHANDLER_FN
37
 INTEGER ERRHANDLER, IERROR
38
 MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
39
 INTEGER COMM, ERRHANDLER, IERROR
40
41
 MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
42
 INTEGER COMM, ERRHANDLER, IERROR
43
 MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
44
 INTEGER ERRHANDLER, IERROR
45
46
 MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
47
 INTEGER ERRORCODE, ERRORCLASS, IERROR
48
```

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR) INTEGER ERRORCODE, RESULTLEN, IERROR	1 2
CHARACTER*(*) STRING	3
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)	4
INTEGER FH, ERRORCODE, IERROR	5 6
	7
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL FILE_ERRHANDLER_FN	8
INTEGER ERRHANDLER, IERROR	9
	10
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	11
INTEGER FILE, ERRHANDLER, IERROR	12
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	13
INTEGER FILE, ERRHANDLER, IERROR	14
	15
MPI_FINALIZED(FLAG, IERROR) LOGICAL FLAG	16
INTEGER IERROR	17
INTEGER TERROR	18
MPI_FINALIZE(IERROR)	19
INTEGER IERROR	20 21
MPI_FREE_MEM(BASE, IERROR)	21
<pre><type> BASE(*)</type></pre>	22
INTEGER IERROR	20 24
	25
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)	26
CHARACTER*(*) VERSION	27
INTEGER RESULTLEN, IERROR	28
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)	29
CHARACTER*(*) NAME	30
INTEGER RESULTLEN, IERROR	31
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)	32
INTEGER VERSION, SUBVERSION, IERROR	33
	34
MPI_INITIALIZED(FLAG, IERROR)	35
LOGICAL FLAG	36
INTEGER IERROR	37
MPI_INIT(IERROR)	38
INTEGER IERROR	39
	40
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)	41
INTEGER WIN, ERRORCODE, IERROR	42
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)	43
EXTERNAL WIN_ERRHANDLER_FN	44 45
INTEGER ERRHANDLER, IERROR	45
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)	40
INTEGER WIN, ERRHANDLER, IERROR	48

```
1
 MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
\mathbf{2}
 INTEGER WIN, ERRHANDLER, IERROR
3
4
 A.5.7 The Info Object Fortran Bindings
5
6
 MPI_INFO_CREATE(INFO, IERROR)
7
 INTEGER INFO, IERROR
8
 MPI_INFO_DELETE(INFO, KEY, IERROR)
9
 INTEGER INFO, IERROR
10
 CHARACTER*(*) KEY
11
12
 MPI_INFO_DUP(INFO, NEWINFO, IERROR)
13
 INTEGER INFO, NEWINFO, IERROR
14
 MPI_INFO_FREE(INFO, IERROR)
15
 INTEGER INFO, IERROR
16
17
 MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
18
 INTEGER INFO, VALUELEN, IERROR
19
 CHARACTER*(*) KEY, VALUE
20
 LOGICAL FLAG
21
 MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
22
 INTEGER INFO, NKEYS, IERROR
23
^{24}
 MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
25
 INTEGER INFO, N, IERROR
26
 CHARACTER*(*) KEY
27
28
 MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
 INTEGER INFO, VALUELEN, IERROR
29
30
 LOGICAL FLAG
^{31}
 CHARACTER*(*) KEY
32
 MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
33
 INTEGER INFO, IERROR
34
 CHARACTER*(*) KEY, VALUE
35
36
37
 A.5.8 Process Creation and Management Fortran Bindings
38
 MPI_CLOSE_PORT(PORT_NAME, IERROR)
39
 CHARACTER*(*) PORT_NAME
40
 INTEGER IERROR
41
42
 MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
43
 CHARACTER*(*) PORT_NAME
44
 INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
45
 MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
46
 CHARACTER*(*) PORT_NAME
47
 INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
48
```

MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR	1 2
	3
MPI_COMM_GET_PARENT(PARENT, IERROR)	4
INTEGER PARENT, IERROR	5
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)	6
INTEGER FD, INTERCOMM, IERROR	7
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,	8
ARRAY_OF_ERRCODES, IERROR)	9
CHARACTER*(*) COMMAND, ARGV(*)	10
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),	11
IERROR	12
	13 14
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,	14
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,	16
ARRAY_OF_ERRCODES, IERROR)	17
INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,	18
INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)	19
CHARACIER*(*) ARRAI_OF_COMMANDS(*), ARRAI_OF_ARGV(COUNI, *)	20
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	21
CHARACTER*(*) SERVICE_NAME, PORT_NAME	22
INTEGER INFO, IERROR	23
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)	24
CHARACTER*(*) PORT_NAME	25
INTEGER INFO, IERROR	26
	27
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	28
INTEGER INFO, IERROR	29
CHARACTER*(*) SERVICE_NAME, PORT_NAME	30
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	31
INTEGER INFO, IERROR	32
CHARACTER*(*) SERVICE_NAME, PORT_NAME	33 34
	35
A.5.9 One-Sided Communications Fortran Bindings	36
A.S.S One Sidea communications Fortrait Dindings	37
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	38
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)	39
<type> ORIGIN_ADDR(*)</type>	40
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	41
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	42
TARGET_DATATYPE, OP, WIN, IERROR	43
MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,	44
TARGET_RANK, TARGET_DISP, WIN, IERROR)	45
<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)</type>	46
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	47
INTEGER DATATYPE, TARGET_RANK, WIN, IERROR	48

```
1
 MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
\mathbf{2}
 TARGET_DISP, OP, WIN, IERROR)
3
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
4
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
5
 INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
6
 MPI GET ACCUMULATE (ORIGIN ADDR, ORIGIN COUNT, ORIGIN DATATYPE, RESULT ADDR,
7
 RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
8
 TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
9
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
10
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
11
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
12
 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
13
14
 MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
15
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
16
 <type> ORIGIN_ADDR(*)
17
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
18
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
19
 TARGET_DATATYPE, WIN, IERROR
20
 MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
21
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
22
 <type> ORIGIN_ADDR(*)
23
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
24
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
25
 TARGET_DATATYPE, WIN, IERROR
26
27
 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
28
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
29
 IERROR)
30
 <type> ORIGIN_ADDR(*)
31
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
32
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
33
 TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
34
 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
35
 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
36
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
37
 IERROR)
38
 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
39
 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
40
 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
41
 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
42
 IERROR
43
^{44}
 MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
45
 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
46
 IERROR)
47
 <type> ORIGIN_ADDR(*)
48
```

1 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP  $\mathbf{2}$ INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, IERROR 4 MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 5 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST, 6 IERROR) 7 <type> ORIGIN_ADDR(*) 8 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 9 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, 10TARGET_DATATYPE, WIN, REQUEST, IERROR 11 MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 1213 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 14INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 15If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 16INTERFACE MPI_WIN_ALLOCATE_SHARED 17SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, & 18 BASEPTR, WIN, IERROR) 19 IMPORT :: MPI_ADDRESS_KIND 20INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 21INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR 22 END SUBROUTINE 23SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, & 24BASEPTR, WIN, IERROR) 25USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 26IMPORT :: MPI_ADDRESS_KIND 27INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 28 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE 29TYPE(C_PTR) :: BASEPTR 30 END SUBROUTINE 31END INTERFACE 32 MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 33 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 34 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 35 If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 36 37 INTERFACE MPI_WIN_ALLOCATE SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & 38 39 WIN, IERROR) 40IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 41 42INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR END SUBROUTINE 43 SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & 44 45WIN, IERROR) 46USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 47IMPORT :: MPI_ADDRESS_KIND 48INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR

```
1
 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
\mathbf{2}
 TYPE(C_PTR) :: BASEPTR
3
 END SUBROUTINE
4
 END INTERFACE
5
 MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
6
 INTEGER WIN, IERROR
7
 <type> BASE(*)
8
 INTEGER (KIND=MPI_ADDRESS_KIND) SIZE
9
10
 MPI_WIN_COMPLETE(WIN, IERROR)
11
 INTEGER WIN, IERROR
12
 MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
13
 <type> BASE(*)
14
 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
15
 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
16
17
 MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
18
 INTEGER INFO, COMM, WIN, IERROR
19
 MPI_WIN_DETACH(WIN, BASE, IERROR)
20
 INTEGER WIN, IERROR
21
 <type> BASE(*)
22
23
 MPI_WIN_FENCE(ASSERT, WIN, IERROR)
24
 INTEGER ASSERT, WIN, IERROR
25
26
 MPI_WIN_FLUSH_ALL(WIN, IERROR)
 INTEGER WIN, IERROR
27
28
 MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
29
 INTEGER WIN, IERROR
30
^{31}
 MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
32
 INTEGER RANK, WIN, IERROR
33
 MPI_WIN_FLUSH(RANK, WIN, IERROR)
34
 INTEGER RANK, WIN, IERROR
35
36
 MPI_WIN_FREE(WIN, IERROR)
37
 INTEGER WIN, IERROR
38
 MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
39
 INTEGER WIN, GROUP, IERROR
40
41
 MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
42
 INTEGER WIN, INFO_USED, IERROR
43
 MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
44
 INTEGER ASSERT, WIN, IERROR
45
46
 MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
47
 INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
48
```

1 MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR) 2 INTEGER GROUP, ASSERT, WIN, IERROR 3 MPI_WIN_SET_INFO(WIN, INFO, IERROR) 4 INTEGER WIN, INFO, IERROR 56 MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP_UNIT, IERROR 7 8 INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 9 If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 10INTERFACE MPI_WIN_SHARED_QUERY 11 SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, & 12BASEPTR, IERROR) 13 IMPORT :: MPI_ADDRESS_KIND 14INTEGER :: WIN, RANK, DISP_UNIT, IERROR 15INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR 16END SUBROUTINE 17SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, & 18 BASEPTR, IERROR) 19 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 20IMPORT :: MPI_ADDRESS_KIND 21INTEGER :: WIN, RANK, DISP_UNIT, IERROR 22 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE 23TYPE(C_PTR) :: BASEPTR 24END SUBROUTINE 25END INTERFACE 2627MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) 28 INTEGER GROUP, ASSERT, WIN, IERROR 29 MPI_WIN_SYNC(WIN, IERROR) 30 INTEGER WIN, IERROR 3132 MPI_WIN_TEST(WIN, FLAG, IERROR) 33 INTEGER WIN, IERROR 34 LOGICAL FLAG 35 MPI_WIN_UNLOCK_ALL(WIN, IERROR) 36 INTEGER WIN, IERROR 37 38 MPI_WIN_UNLOCK(RANK, WIN, IERROR) 39 INTEGER RANK, WIN, IERROR 40 MPI_WIN_WAIT(WIN, IERROR) 41 INTEGER WIN, IERROR 4243 44A.5.10 External Interfaces Fortran Bindings 45MPI_GREQUEST_COMPLETE(REQUEST, IERROR) 4647INTEGER REQUEST, IERROR 48

```
1
 MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
\mathbf{2}
 IERROR)
3
 INTEGER REQUEST, IERROR
4
 EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
\mathbf{5}
 INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
6
 MPI INIT THREAD (REQUIRED, PROVIDED, IERROR)
7
 INTEGER REQUIRED, PROVIDED, IERROR
8
9
 MPI_IS_THREAD_MAIN(FLAG, IERROR)
10
 LOGICAL FLAG
11
 INTEGER IERROR
12
 MPI_QUERY_THREAD(PROVIDED, IERROR)
13
 INTEGER PROVIDED, IERROR
14
15
 MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
16
 INTEGER STATUS(MPI_STATUS_SIZE), IERROR
17
 LOGICAL FLAG
18
 MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
19
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
20
21
 MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
22
 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
23
 INTEGER (KIND=MPI_COUNT_KIND) COUNT
^{24}
25
 A.5.11 I/O Fortran Bindings
26
27
 MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION,
28
 EXTRA_STATE, IERROR)
29
 <TYPE> USERBUF(*), FILEBUF(*)
30
 INTEGER COUNT, DATATYPE, IERROR
31
 INTEGER(KIND=MPI_OFFSET_KIND) POSITION
32
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
33
34
 MPI_FILE_CLOSE(FH, IERROR)
 INTEGER FH, IERROR
35
36
 MPI_FILE_DELETE(FILENAME, INFO, IERROR)
37
 CHARACTER*(*) FILENAME
38
 INTEGER INFO, IERROR
39
40
 MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
41
 INTEGER FH, AMODE, IERROR
42
 MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
43
 INTEGER FH, IERROR
44
 LOGICAL FLAG
45
 MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
46
47
 INTEGER FH, IERROR
48
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
```

MPI_FILE_GET_GROUP(FH, GROUP, IERROR) INTEGER FH, GROUP, IERROR	1 2
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)	3
INTEGER FH, INFO_USED, IERROR	4
	5
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	7
INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	8
	9
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	10
INTEGER FH, IERROR	11
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	12
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	13
INTEGER FH, IERROR	14
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	15
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)	16 17
INTEGER FH, DATATYPE, IERROR	18
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT	19
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)	20
INTEGER FH, ETYPE, FILETYPE, IERROR	21
CHARACTER*(*) DATAREP	22
INTEGER(KIND=MPI_OFFSET_KIND) DISP	23
MOT ETLE TREAD ALL (FIL DILE COUNT DATATVDE DECHECT TEDDOD)	24
<pre>MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)</pre>	25
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	26 27
	21
MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	29
<pre><type> BUF(*) INTEGED EU COUNT DATATYDE DECUEGT LEDDOD</type></pre>	30
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	31
	32
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	33
<type> BUF(*)</type>	34
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	35
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	36 37
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	38
<type> BUF(*)</type>	39
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	40
MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	41
<type> BUF(*)</type>	42
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	43
MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	44
<pre><type> BUF(*)</type></pre>	45
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	46 47
MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	47 48
In I_IILL_IWATIL_AI_ALL(IN, OFFEL, BOF, COONI, DATAIILE, ALQUESI, IERAUA)	10

1 <type> BUF(*)  $\mathbf{2}$ INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 3 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 4 MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR) 5<type> BUF(*) 6 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 7 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 8 9 MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 10<type> BUF(*) 11 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 12MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 13 <type> BUF(*) 14INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 1516MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) 17CHARACTER*(*) FILENAME 18 INTEGER COMM, AMODE, INFO, FH, IERROR 19 MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) 20INTEGER FH, IERROR 21INTEGER(KIND=MPI_OFFSET_KIND) SIZE 2223MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)  24 <type> BUF(*) 25INTEGER FH, COUNT, DATATYPE, IERROR 26MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR) 27<type> BUF(*) 28INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 2930 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)  31 <type> BUF(*) 32 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 33 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 34 <type> BUF(*) 35 INTEGER FH, COUNT, DATATYPE, IERROR 36 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 37 38MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR) 39 <type> BUF(*) 40INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 41 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 42<type> BUF(*) 43 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 44INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 4546MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 47<type> BUF(*) 48

INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	1 2
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*)</type>	3 4
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	5 6
<pre>MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)</pre>	7 8 9
INTEGER FH, COUNT, DATATYPE, IERROR	9 10
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR) <type> BUF(*)</type>	11 12
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	13 14
<pre>MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	14
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	16 17
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	18
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>	19 20
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)	20 21
INTEGER FH, WHENCE, IERROR	22
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	23 24
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)	25
INTEGER FH, WHENCE, IERROR	26
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	27 28
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)	28 29
INTEGER FH, IERROR LOGICAL FLAG	30
	31
MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR	32 33
	34
MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR	35
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	36
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)	37 38
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR	39
CHARACTER*(*) DATAREP	40
INTEGER(KIND=MPI_OFFSET_KIND) DISP	41
MPI_FILE_SYNC(FH, IERROR)	42
INTEGER FH, IERROR	43 44
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	45
<type> BUF(*)</type>	46
INTEGER FH, COUNT, DATATYPE, IERROR	47
	48

```
1
 MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
\mathbf{2}
 <type> BUF(*)
3
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
4
 MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
5
 <type> BUF(*)
6
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
7
8
 MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
9
 <type> BUF(*)
10
 INTEGER FH, COUNT, DATATYPE, IERROR
11
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
12
 MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
13
 <type> BUF(*)
14
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
15
16
 MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
17
 <type> BUF(*)
18
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
19
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
20
 MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
21
 <type> BUF(*)
22
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
23
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
^{24}
25
 MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
26
 <type> BUF(*)
27
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
28
 MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
29
 <type> BUF(*)
30
 INTEGER FH, COUNT, DATATYPE, IERROR
^{31}
32
 MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
33
 <type> BUF(*)
34
 INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
35
 MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
36
 <type> BUF(*)
37
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
38
39
 MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
40
 <type> BUF(*)
41
 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
42
 MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
43
 DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
44
 CHARACTER*(*) DATAREP
45
 EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
46
 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
47
 INTEGER IERROR
48
```

A.5.12 Language Bindings Fortran Bindings	1
MPI_F_SYNC_REG(buf)	2 3
<type> buf(*)</type>	4
MPI_SIZEOF(X, SIZE, IERROR)	5
<type> X</type>	6
INTEGER SIZE, IERROR	7 8
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)	9
TYPE(MPI_Status) :: F08_STATUS	10
INTEGER :: F_STATUS(MPI_STATUS_SIZE)	11
INTEGER IERROR	12
MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)	13
INTEGER :: F_STATUS(MPI_STATUS_SIZE)	14
TYPE(MPI_Status) :: F08_STATUS	15 16
INTEGER IERROR	10
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)	18
INTEGER P, R, NEWTYPE, IERROR	19
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)	20
INTEGER R, NEWTYPE, IERROR	21
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)	22
INTEGER P, R, NEWTYPE, IERROR	23 24
	24 25
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)	26
INTEGER TYPECLASS, SIZE, DATATYPE, IERROR	27
	28
A.5.13 Tools / Profiling Interface Fortran Bindings	29
MPI_PCONTROL(LEVEL)	30
INTEGER LEVEL	31
	32 33
A E 14 Desugested Fasture Diadians	34
A.5.14 Deprecated Fortran Bindings	35
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)	36
INTEGER COMM, KEYVAL, IERROR	37
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	38
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	39
LOGICAL FLAG	40 41
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)	41
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	43
MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	44
ATTRIBUTE_VAL_OUT, FLAG, IERR)	45
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	46
ATTRIBUTE_VAL_OUT, IERR	47
	48

1	LOGICAL FLAG
2 3	MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
4	EXTERNAL COPY_FN, DELETE_FN
5	INTEGER KEYVAL, EXTRA_STATE, IERROR
6	MPI_KEYVAL_FREE(KEYVAL, IERROR)
7	INTEGER KEYVAL, IERROR
8	MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
9	ATTRIBUTE_VAL_OUT, FLAG, IERR)
10 11	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
12	ATTRIBUTE_VAL_OUT, IERR
13	LOGICAL FLAG
14	MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
15	INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
16	SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
17 18	ATTRIBUTE_VAL_OUT, FLAG, IERR)
19	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
20	ATTRIBUTE_VAL_OUT, IERR
21	LOGICAL FLAG
22	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
23 24	INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
24 25	
26	
27	
28	
29	
30 31	
32	
33	
34	
35	
36 27	
37 38	
39	
40	
41	
42	
43 44	
44 45	
46	
47	
48	

## Annex B

## Change-Log

Annex B.1 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. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.5 were already introduced in the corresponding sections in previous versions of this standard.

#### B.1 Changes from Version 3.1 to Version 3.2

#### B.1.1 Changes in MPI-3.2

 Section 2.4, 3.4, 3.7.2, 3.7.3, 3.8.1, 3.8.2, 3.8.3, 5.13, 13.4.5, and Annex A.2 on pages 11, 39, 50, 54, 67, 70, 72, 217, 554, and 719.
 The semantic terms were updated and the new Annex A.2 summarizes the semantics of all communicating MPI routines. The definition of the MPI semantic term nonblocking, in particular, has been clarified. The *de jure* meaning before MPI-4.0 was

equivalent to the MPI-4.0 meaning of the MPI semantic term incomplete. However, the *de facto* usage of the term nonblocking, when applied to MPI procedure calls, has always been the combination of incomplete and local. Consequently, the definition of the MPI semantic term blocking has been clarified from being equivalent to complete only to being equivalent to complete or non-local (or both). The definition of a collective procedure was also clarified in the way that the nonblocking initiation routines of collective operations are now also *de jure* collective.

- 2. Section 3.8.4 on page 74.
  Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.
  3. Sections 3.7.3, 3.9, 5.13, 7.8, and 7.9 on pages 54, 76, 217, 349, and 354.
- Persistent collective communication and persistent neighborhood communication are added to the standard.
- 4. Section 6.4.2 on page 257, and MPI-3.1 Section 6.4.2 on page 237. The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer

 31 

 $46 \\ 47$ 

1		propagate info hints. This change may affect backward compatibility.
3 4 5 6 7	5.	Sections 6.4.4, 11.2.7, and 13.2.8 on pages 269, 441, and 526, and MPI-3.1 Sections 6.4.4, 11.2.7, and 13.2.8 on pages 248, 415, and 500. The definition of info hints was updated to allow applications to provide assertions regarding their usage of MPI objects and operations.
8 9 10 11 12	6.	Section 6.4.4 on page 269. The new info hints mpi_assert_no_any_tag, mpi_assert_no_any_source, mpi_assert_exact_length, and mpi_assert_allow_overtaking were added for use with commu- nicators.
13 14	7.	Section 6.4.2 on page 257. The MPI_COMM_IDUP_WITH_INFO function was added.
15 16 17 18 19	8.	Sections 6.4.4, 11.2.7, and 13.2.8 on pages 269, 441, and 526. The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.
20 21 22	9.	Section 14.3.9 and Table 14.5 on pages 622 and 624. MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return MPI_T_ERR_INVALID_INDEX instead of MPI_T_ERR_INVALID_ITEM.
23 24 25 26	10.	Section 7.5. MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of di- mensions passed to the routine is set to 0 and the number of nodes is set to 1.
27 28 29 30 31	11.	Sections 2.8, 8.3, 8.5, and 8.7 on pages 22, 366, 374, and 381. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF instead of MPI_COMM_WORLD. The definition of MPI_ERRORS_ARE_FATAL was clarified to cover all connected processes, and a new error handler, MPI_ERRORS_ABORT, was created to limit the scope of aborting.
32 33 34 35 36	12.	Section 12.3 on page 509 The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 binding for MPI_SET_STATUS_CANCELLED marked as INTENT(OUT). It has been fixed to be INTENT(IN).
37 38 39 40	13.	Sections 8.3 and 8.4 on pages 366 and 374. Clarified definition of errors to say that MPI should continue whenever possible and allow the user to recover from errors.
41 42	B.2	Changes from Version 3.0 to Version 3.1
43 44	B.2.3	I Fixes to Errata in Previous Versions of MPI
45 46 47 48	1.	Chapters 3–18, Annex A.4 on page 746, and Example 5.21 on page 189, and MPI-3.0 Chapters 3-17, Annex A.3 on page 707, and Example 5.21 on page 187. Within the mpi_f08 Fortran support method, BIND(C) was removed from all SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions.

2.	Section 3.2.5 on page 32, and MPI-3.0 Section 3.2.5 on page 30. The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived type TYPE(MPI_Status) must be of type INTEGER.	1 2 3
3.	Section 3.8.2 on page 70, and MPI-3.0 Section 3.8.2 on page 67. The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incorrectly defined as INTEGER (instead as LOGICAL).	4 5 6 7
4.	Section 6.4.2 on page 257, and MPI-3.0 Section 6.4.2 on page 237. In the mpi_f08 binding of MPI_COMM_IDUP, the output argument newcomm is declared as ASYNCHRONOUS.	8 9 10 11
5.	Section 6.4.4 on page 269, and MPI-3.0 Section 6.4.4 on page 248. In the mpi_f08 binding of MPI_COMM_SET_INFO, the intent of comm is IN, and the optional output argument ierror was missing.	12 13 14
6.	Section 7.6 on page 334, and MPI-3.0 Sections 7.6, on pages 314. In the case of virtual general graph topolgies (created with MPI_CART_CREATE), the use of neighborhood collective communication is restricted to adjacency matrices with the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix).	15 16 17 18 19 20
7.	Section 8.1.1 on page 359, and MPI-3.0 Section 8.1.1 on page 335. In the mpi_f08 binding of MPI_GET_LIBRARY_VERSION, a typo in the resultlen argument was corrected.	21 22 23 24
8.	Sections 8.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 11.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 11.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 11.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR), 14.2.1 and 14.2.7 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.	25 26 27 28 29 30
9.	Section 11.2.1 on page 429, and MPI-3.0 Section 11.2.2 on page 407. The same_size info key can be used with all window flavors, and requires that all processes in the process group of the communicator have provided this info key with the same value.	31 32 33 34 35
10.	Section 11.3.4 on page 449, and MPI-3.0 Section 11.3.4 on page 424. Origin buffer arguments to MPI_GET_ACCUMULATE are ignored when the MPI_NO_OP operation is used.	36 37 38
11.	Section 11.3.4 on page 449, and MPI-3.0 Section 11.3.4 on page 424. Clarify the roles of origin, result, and target communication parameters in MPI_GET_ACCUMULATE.	39 40 41 42
12.	Section 14.3 on page 593, and MPI-3.0 Section 14.3 on page 561 New paragraph and advice to users clarifying intent of variable names in the tools information interface.	43 44 45 46
13.	Section 14.3.3 on page 595, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.	47 48

2 3 4 5 6	A. Sections 14.3.6, 14.3.7, and 14.3.8 on pages 599, 606, and 618, and MPI-3.0 Sections 14.3.6, 14.3.7, and 14.3.8 on pages 567, 573, and 584. In functions MPI_T_CVAR_GET_INFO, MPI_T_PVAR_GET_INFO, and MPI_T_CATEGORY_GET_INFO, clarification of parameters that must be identical for equivalent control variable / performance variable / category names across connected processes.
7 8 15 9	5. Section 14.3.7 on page 606, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.
10 11 12	5. Section 14.3.7 on page 606, and MPI-3.0 Section 14.3.7 on page 579, line 7. Clarify the return code when bad handle is passed to an MPI_T_PVAR_* routine.
14 15	Y. Section 18.1.4 on page 639, and MPI-3.0 Section 17.1.4 on page 603. The advice to implementors at the end of the section was rewritten and moved into the following section.
16 17 18 19	3. Section 18.1.5 on page 640, and MPI-3.0 Section 17.1.5 on page 605. The section was fully rewritten. The linker name concept was substituted by defining specific procedure names.
²⁰ 19 ²¹ 22	2. Section 18.1.6 on page 645, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.
24 25 26	D. Annexes A.3, A.4, and A.5 on pages 722, 746, and 801, and MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756. The predefined callback MPI_CONVERSION_FN_NULL was added to all three annexes.
27 28 21 29 30 31	. Annex A.4.4 on page 767, and MPI-3.0 Annex A.3.4 on page 724. In the mpi_f08 binding of MPI_{COMM TYPE WIN}_{DUP NULL_COPY NULL_DELETE}_FN, all INTENT() information was removed.
32 33 B.2	.2 Changes in MPI-3.1
34 35 36 37	. Sections 2.6.4 and 4.1.5 on pages 21 and 103. The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can be implemented as macros.
<ol> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ol>	2. Sections 8.1.1, 8.7, and 12.4 on pages 359, 381, and 510. The routines MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and MPI_GET_LIBRARY_VERSION are callable from threads without restriction (in the sense of MPI_THREAD_MULTIPLE), irrespective of the actual level of thread support provided, in the case where the im- plementation supports threads.
44 45 46 47 48	3. Section 11.2.1 on page 429. The same_disp_unit info key was added for use in RMA window creation routines.

4.	Sections 13.4.2 and 13.4.3 on pages 535 and 540. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL,	1 2 3
	MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL	3 4
5.	Sections 14.3.6, 14.3.7, and 14.3.8 on pages 599, 606, and 618.	5
	Clarified that NULL parameters can be provided in	6
	MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.	7
0		8
6.	Sections 14.3.6, 14.3.7, 14.3.8, and 14.3.9 on pages 599, 606, 618, and 622.	9
	New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX,	10
	MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of vari-	11
	ables and categories. The error codes MPI_T_ERR_INVALID and	12
	MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface.	13
		14
B.3	Changes from Version 2.2 to Version 3.0	15
		16
B.3.1	Fixes to Errata in Previous Versions of MPI	17
1.	Sections 2.6.2 and 2.6.3 on pages 20 and 21, and MPI-2.2 Section 2.6.2 on page 17,	18
	lines 41-42, Section 2.6.3 on page 18, lines 15-16, and Section 2.6.4 on page 18, lines 40-	19
	41.	20
	This is an MPI-2 erratum: The scope for the reserved prefix MPI_ and the C++	21
	namespace MPI is now any name as originally intended in MPI-1.	22
-		23
2.	Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 27, 178, 566, and	24
	697, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and	25
	Annex A.1.1 on pages 27, 164, 433, 472 and 513	26
	This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL,	27
	MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and	28
	MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding	29
	to the C++ types bool, std::complex <float>, std::complex<double>, and</double></float>	30 31
	std::complex <long double="">. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX,</long>	31
	and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-	33
	standard $C++$ types Complex<> were substituted by the standard types	34
	std::complex<>.	35
	Dodcomptox	36
3.	Sections $5.9.2$ on pages 178 and MPI-2.2 Section $5.9.2$ , page 165, line 47.	37
	This is an MPI-2.2 erratum: $MPI_C_COMPLEX$ was added to the "Complex" reduc-	38
	tion group.	39
4	Section 7.5.5 on page 222 and MDI 2.2 Section 7.5.5 on page 257 C++ interface on	40
4.	Section 7.5.5 on page 322, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on	41
	page 264, line 3. This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are	42
	now defined as int& indegree and int& outdegree in the C++ interface of	43
	MPI_DIST_GRAPH_NEIGHBORS_COUNT.	44
		45
5.	Section 13.5.2, Table 13.2 on page 566, and MPI-2.2, Section 13.5.3, Table 13.2 on	46
	page 433.	47
		48

1 2	This was an MPI-2.2 erratum: 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 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.</li> </ol>
7 , 8 9 10 11 12	<ul> <li>7. Annex A.1.1 on page 697, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37-41. This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.</li> </ul>
¹³ ₁₄ B.3	.2 Changes in MPI-3.0
16 17 18	1. Section 2.6.1 on page 19, Section 16.2 on page 630 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.3.1 on page 839 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility.
19         20         21         22         23         24         25         26         27         28	2. Section 2.6.1 on page 19, Section 15.1 on page 625 and Section 16.1 on page 629. 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.
29 30 31 32	3. Section 2.3 on page 10. Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.
	I. Section 2.5.4 on page 16 and Section 7.5.4 on page 316. 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.
37 38 39 40	5. Section 2.5.4 on page 16 and Section 8.1.1 on page 359. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.
41 42 43 44 45 46 47 48	5. Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 18, 27, 29, 178, Sections 4.1, 4.1.7, 4.1.8, 4.1.11, 12.3 on pages 85, 108, 110, 113, 508, and Annex A.1.1 on page 697. 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 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.

- 3 7. Chapter 3 on page 25 until Chapter 18 on page 633. 4 In the C language bindings, the array-arguments' interfaces were modified to consis-5tently use use [] instead of *. 6 Exceptions are MPI_INIT, which continues to use char ***argv (correct because of  $\overline{7}$ subtle rules regarding the use of the & operator with char *argv[]), and 8 MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT. 9 10 8. Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 32, 103, 113, 134. 11 The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the 12count argument to MPI_UNDEFINED when that argument would overflow. The func-13 tions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument 14 to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 rou-15tines, the type and semantics of the count arguments remain unchanged, i.e., int or 16INTEGER. 1718
- 9. Section 3.2.6 on page 34, and Section 3.8 on page 66. MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.
- Section 3.8 on page 66 and Section 3.11 on page 83. 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 (i.e., the new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.
- 11. Sections 3.8.2, 3.8.3, 18.2.4, A.1.1 on pages 70, 72, 682, 697. Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message with the new routines MPI_MRECV and MPI_IMRECV regardless of other intervening probe or receive operations. The opaque object MPI_Message, the 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 87 and Section 4.1.13 on page 118. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.
- Chapter 5 on page 143 and Section 5.12 on page 199.
   Added nonblocking interfaces to all collective operations.
- Sections 6.4.2, 6.4.4, 11.2.7, on pages 257, 269, 441.
   The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.
- 15. Section 6.4.2 on page 257. Added MPI_COMM_IDUP.

1

2

19

20

21

22

23

24 25

26

27

28

29

30

 31 

32

33

34 35

36

37

38

39

40 41

42

43

44

 $45 \\ 46$ 

47

1 2 3 4	16.	Section 6.4.2 on page 257. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.
5 6 7 8	17.	Section 6.4.2 on page 257. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- stant MPI_COMM_TYPE_SHARED.
9 10 11 12 13	18.	Section 6.6.2 on page 281. 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.
14 15 16	19.	Section 6.8 on page 302. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.
17 18 19	20.	Section 7.5.8 on page 332. MPI_CART_MAP can also be used for a zero-dimensional topologies.
20 21 22 23 24 25 26 27 28 29 30	21.	Section 7.6 on page 334 and Section 7.7 on page 343. 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_ALLTOALLV, 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_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and 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 34 35	22.	Section 8.7 on page 381 and Section 12.4.3 on page 513. The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After MPI is initialized, the application can access information about the execution envi- ronment by querying the new predefined info object MPI_INFO_ENV.
36 37	23.	Section 8.7 on page 381. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.
38 39 40 41 42	24.	Chapter 11 on page 427. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes.
43 44	25.	Section 14.3 on page 593. A new MPI Tool Information Interface was added.
45 46		The following changes are related to the Fortran language support.
47 48	26.	Section 2.3 on page 10, and Sections 18.1.1, 18.1.2, 18.1.7 on pages 633, 634, and 649. The new mpi_08 Fortran module was introduced.

- 27. Section 2.5.1 on page 14, and Sections 18.1.2, 18.1.3, 18.1.7 on pages 634, 637, and 649. 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 16, 18, Sections 18.1.1, 18.1.10, 18.1.11, 18.1.12, 18.1.13 on pages 633, 659, 661, 661, 664, and Sections 18.1.2, 18.1.3, 18.1.7 on pages 634, 637, 649.

Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [41], and the compile-time constant  ${\tt MPI_SUBARRAYS_SUPPORTED \ was \ set \ to \ . \ TRUE.. \ With \ this, \ Fortran \ subscript \ triplets}$ 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 20, Section 18.1.2 on page 634, and Section 18.1.7 on page 649. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.
- 30. Section 3.2.5 on page 32, Sections 18.1.2, 18.1.3, 18.1.7, on pages 634, 637, 649, and Section 18.2.5 on page 684. Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined. New conversion routines were added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new type MPI_F08_status, and the external variables MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE were added.
- 31. Section 3.6 on page 46.

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 85, Section 4.1.6 on page 106, and Section 18.1.15 on page 665. The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in C/C++) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI_TYPE_CREATE_RESIZED to this datatype handle.
- 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, 18.1.9 on pages 113, 41 185, 191, 296, 302, 368, 370, 371, 625, and 651. In some routines, the dummy ar-42gument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the 43mpi and mpi_08 modules guarantee keyword-based actual argument lists. The ar-44gument name type was changed in MPI_TYPE_DUP, the Fortran USER_FUNCTION of MPI_OP_CREATE, MPI_TYPE_SET_ATTR, MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME,

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

1516

17

18 19

20

21

22

23 24 

25

26

2728

29

30

 31 32

33

34

35

36

37

38

39

40

4546

1 2 3 4 5 6 7	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.
8 9 <b>3</b> 10 11 12	<ol> <li>Section 6.7.2 on page 288.</li> <li>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.</li> </ol>
$13 \\ 14 \\ 15 \\ 16 \\ 17$	5. Section 8.2 on page 363. With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a non-standard Cray-pointer.
19 20	<ol> <li>Section 18.1.15 on page 665, and Section 18.1.7 on page 649.</li> <li>Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.</li> </ol>
21 22 3 23 24 25 26 27 28 29	7. Section 18.1.16 on page 667 to Section 18.1.19 on page 676, Section 18.1.7 on page 649, and Section 18.1.8 on page 650. The sections about Fortran optimization problems and their solutions 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 semantics of the ASYNCHRONOUS attribute is 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 18.1.7.
$egin{array}{ccc} 30 & & & 3 \ 31 & & & 3 \ 32 & & & & \ 33 & & & & \end{array}$	<ol> <li>Section 18.1.2 on page 634.</li> <li>Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.</li> </ol>
34 <b>3</b> 35	9. Section 18.1.3 on page 637, and Section 18.1.7 on page 649. The existing mpi Fortran module must implement compile-time argument checking.
36 37 4 38	). Section 18.1.4 on page 639. The use of the mpif.h Fortran include file is now strongly discouraged.
³⁹ 4 40 41 42 43 44	<ol> <li>Section A.1.1, Table "Predefined functions" on page 705, Section A.1.3 on page 712, and Section A.4.4 on page 767.</li> <li>Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI) that have the BIND(C) attribute; userwritten callbacks must be modified if the mpi_f08 module is used.</li> </ol>
45 4 46 47 48	<ol> <li>Section A.1.3 on page 712.</li> <li>In some routines, the Fortran callback prototype names were changed fromFN toFUNCTION to be consistent with the other language bindings.</li> </ol>

B.4	Changes from Version 2.1 to Version 2.2	1
1.	Section 2.5.4 on page 16.	2 3
	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
2.	Section 2.6 on page 19, and Section 16.2 on page 630.	7
	The C++ language bindings have been deprecated and may be removed in a future	8
	version of the MPI specification.	9 10
3.	Section $3.2.2$ on page $27$ .	11
-	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 27.	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 $MPI$ data types.	21
5.	Section 3.4 on page 39, Section 3.7.2 on page 50, Section 3.9 on page 76, and Section 5.1	22
	on page 143.	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 to in progress.	27
6.	Section 3.7 on page 49.	28
	The Advice to users for IBSEND and IRSEND was slightly changed.	29
-		30
7.	Section 3.7.3 on page 54.	31
	The advice to free an active request was removed in the Advice to users for	32
	MPI_REQUEST_FREE.	33
8	Section $3.7.6$ on page $66$ .	34
0.	MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.	35
	Win _ NEQUEST_GET_STATUS changed to permit macrive of num requests as input.	36
9.	Section 5.8 on page 170.	37
	"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and	38
	MPI_ALLTOALLW for intracommunicators.	39
		40
10.	Section $5.9.2$ on page 178.	41
	Predefined parameterized datatypes (e.g., returned by	42
	MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g.	43
	MPI_REAL8) have been added to the list of valid datatypes in reduction operations.	44
11		45
11.	Section 5.9.2 on page 178.	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 191. The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.
8 9 10 11	13.	Section 5.10.1 on page 193. The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan- dard.
12 13	14.	Section 5.11.2 on page 196. Added in place argument to MPI_EXSCAN.
14 15 16 17 18 19 20	15.	Section 6.4.2 on page 257, and Section 6.6 on page 278. 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 257. 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 316. 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 322. 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 322. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
39 40	20.	Section 8.1.1 on page 359. The subversion number changed from 1 to 2.
41 42 43 44	21.	Section 8.3 on page 366, Section 15.2 on page 628, and Annex A.1.3 on page 712. 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 387. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Imple- mentors 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 449. 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		
~ .		8		
24.	Section 12.2 on page 501.	9		
	Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for	10		
	MPI::Grequest::Start() for consistency with the rest of MPI functions that take function	11		
	pointer arguments.	12		
25.	Section $13.5.2$ on page 564, and Table $13.2$ on page 566.	13		
-	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX,	14		
	MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX,	15		
	MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes	16		
	in the external 32 representation.	17		
	-	18		
26.	Section 18.2.7 on page 689.	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 18.13, 18.14, and 18.15 on pages 690-691	22		
	explicitly detailing cross-language attribute behavior. Implementations that matched	23		
	the behavior of the old example will need to be updated.	24 25		
27.	Annex A.1.1 on page 697.	23 26		
	Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 710).	20		
		28		
28.	Annex A.1.1 on page 697. 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.5	Changes from Version 2.0 to Version 2.1	34		
		35		
1.	Section 3.2.2 on page 27, and Annex A.1 on page 697.	36		
	In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn-	37		
	onym for MPI_LONG_LONG_INT.	38		
2	Section 3.2.2 on page 27, and Annex A.1 on page 697.	39		
2.	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.	42		
		43		
3.	Section 3.2.5 on page 32.	44		
	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 85. 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 140. 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 189. 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 248. 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 286. 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 302. 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 Fortran, 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 310. About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must have identical values on all processes of the group of comm_old.
<ul> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ul>	11.	Section 7.5.1 on page 312. 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 314. In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$ , then MPI_COMM_NULL is returned in all processes.

13.	Section 7.5.3 on page 314. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.	1 2 3 4 5 6
	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
14.	Section 7.5.5 on page 322. 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.	10 11 12 13 14
15.	Section 7.5.5 on page 322. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol- ogy, coord is not significant and 0 is returned in rank.	15 16 17 18
16.	Section 7.5.5 on page 322. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.	19 20 21
17.	Section 7.5.6 on page 330. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.	22 23 24 25 26 27
18.	Section 7.5.7 on page 331. 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.	28 29 30 31
18.1.	Section $8.1.1$ on page $359$ . The subversion number changed from 0 to 1.	32 33 34
19.	Section 8.1.2 on page 360. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.	35 36 37 38 39 40
20.	Section 8.3 on page 366. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.	41 42 43 44 45 46 47 48

1 2 3 4 5	21.	Section 8.7 on page 381, 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 423.
6 7 8	22.	Section 8.7 on page 381. 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 24 25	23.	Section 9 on page 391. 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 443. 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 443. 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 449. 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 526. 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 526. 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 530. 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 564. 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 18.1.9 on page 651. About MPI_TYPE_CREATE_F90_XXX:	10 11
	Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of $(XXX,p,r)$ . The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same ( REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination ( p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash- table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (End of advice to implementors.)	12 13 14 15 16 17 18 19 20 21 22
	Section A.1.1 on page 697. MPI_BOTTOM is defined as void * const MPI::BOTTOM.	24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
		44 45 46
		47 48

## Bibliography

- V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1
- [4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.
   1.2
- [5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990. 1.2
- [6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. IJHPCN, 1(1/2/3):91–99, 2004. 11.7
- [7] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. 13.1
- [8] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992. 1.2
- [9] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2
- [10] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. 1.2
- [11] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormholerouted networks for interprocessor communication in multicomputers. In *Proceedings* of the 1990 Supercomputing Conference, pages 647–656, 1990. 7.1

9 10

11

12

13 14

15

16

17 18

19

20

21

22

23 24

25

26

27 28

29

30

 31 

32

33 34

35

36 37

38

39

40

41

42

43

 $\frac{44}{45}$ 

46

47

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 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

- [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. 18.1.3
- [29] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 3.8.1, 3.8.2
- [30] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12
- [31] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6
- [32] T. Hoefler and A. Lumsdaine. Message progression in parallel computing to thread or not to thread? In Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society, Oct. 2008. 5.12
- [33] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07.* IEEE Computer Society/ACM, Nov. 2007. 5.12
- [34] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12
- [35] T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 7.6
- [36] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI common practice, issues, and extensions. In Cotronis et al. [14], pages 345–355. 6.4.2
- [37] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2
- [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

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. Information technology – Programming languages – Fortran – Part 1: Base language, November 2010. 18.1.1, 18.1.2
8 9 10 11 12 13	[41]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TS 29113. TS on further interoperability with C, 2012. http://www.nag.co.uk/sc22wg5/, successfully balloted DTS at ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf. 18.1.1, 18.1.1, 18.1.2, 18.1.7, 28
14 15	[42]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 4.1.4
16 17 18 19	[43]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994 Symposium on Operating Systems Design and Implementation</i> , pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1
20 21 22	[44]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 7.1
23 24 25 26 27	[45]	S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at http://www.netbsd.org/Documentation/lite2/psd/. 10.5.5
28 29	[46]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2
30 31 32	[47]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1
33 34	[48]	William J. Nitzberg. Collective Parallel $I/O$ . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1
35 36 37	[49]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 10.5.5
38 39 40 41	[50]	Paul Pierce. The NX/2 operating system. In <i>Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications</i> , pages 384–390. ACM Press, 1988. 1.2
42 43 44	[51]	Martin Schulz and Bronis R. de Supinski. P ^N MPI tools: A whole lot greater than the sum of their parts. In $ACM/IEEE$ Supercomputing Conference (SC), pages 1–10. ACM, 2007. 14.2.8
45 46 47 48	[52]	K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In <i>Proceedings of Supercomputing '95</i> , December 1995. 13.1

- [53] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. 1.2, 6.1.2
- [54] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2
- [55] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. 6.1
- [56] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, April 1994. http://www.erc.msstate.edu/mpi/mpix.html. 5.2.2
- [57] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994. 6.1.2, 6.5.6
- [58] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996. 13.1
- [59] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2
- [60] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

 24 

General Index

10

11

This index lists mainly terms of the MPI specification. The underlined page numbers refer to the definitions or parts of the definition of the terms. Bold face numbers mark section titles.

12absolute addresses, 18, 103, 669 access epoch, 46313 action 14in function names, 10 15active, 54, 307 16active target communication, 462 17addresses, 117 18 absolute, 18, 103, 669 19correct use, 117 relative displacement, 18, 103 20all-reduce, 189 21nonblocking, 213 22 persistent, 230 23all-to-all, 170 24nonblocking, 209 25persistent, 226 26array arguments, 16 27assertions, 476 ASYNCHRONOUS 28 Fortran attribute, 671 29 attribute, <u>245</u>, <u>286</u>, **689** 30 caching, 244  31 32 backward incompatibilies, 631 33 barrier synchronization, 149 34nonblocking, 201 persistent, 219 35 blocking, 39, 42, 533 36 I/O, 534 37 blocking operation, 1238 blocking procedure, 1239 bounds of datatypes, 108 40broadcast, 150 41 nonblocking, 202 persistent, 219 42buffer allocation, 46 43 buffered, 39, 49, 50 44 nonblocking, 5045buffered send, 424647 $\mathbf{C}$ language binding, 21 48

caching, 243, 244, 286 callback functions language interoperability, 688 prototype definitions, 712 deprecated, 717 cancel, 20, 55, <u>66</u>, **74**, 385, 628, 835 canonical pack and unpack, 140 Cartesian topology, <u>310</u>, **312** change-log, 835 choice, 18 class in function names, 10 clock synchronization, 362 collective, 533 collective communication, 143 correctness. 234 file data access operations, 551 neighborhood, 334 nonblocking, 199 collective operations, 13 collective procedure, 13 commit, 111 COMMON blocks, 674 communication, 427collective, 143 modes, 39 one-sided, 427 point-to-point, 25 RMA, 427 communicator, <u>29</u>, **243**, <u>244</u> complete procedure, 11 Completion, 11completion, 54 multiple, 59 connected, 423 constants, 16, 693, 697 context, 243, 244, 246 control variables tools interface, **599** conversion, 37

#### General Index

counts, 18create in function names, 10 data, 27 data conversion, 37datatypes, 85, 687 delete in function names, 10 deprecated interfaces, 19, 625 derived datatype, <u>13</u>, **85**, **665** disconnected, 423 displacement, 517, 530 distributed graph topology, <u>**310</u></u>, <b>316**</u> dynamically attached memory, 436 elementary datatype, <u>517</u>, 531 empty, 54end of file, 519envelope, 25, 29 environmental inquiries, 360 equivalent datatypes, 14 error handling, 22, 366 error codes and classes, 374 error handlers, 368, 374, 688 I/O, 581 one-sided communication, 478 program error, 22resource error, 22establishing communication, 411 etype, <u>517</u>, 531 exception, 366 exclusive scan nonblocking, 217 persistent, 234 explicit offsets,  $\underline{533}$ ,  $\underline{535}$ exposure epoch, 463extent of datatypes, <u>86</u>, <u>107</u>, **108** true extent, 110 external32 file data representation, 562extra-state, 693 fairness, 44, <u>64</u> file, 517data access, 533 collective operations, 551 explicit offsets, **535** individual file pointers, 540 seek, **552** shared file pointers, 548 split collective, 554 end of file, 519filetype, 518

handle, $519$	1
interoperability, <b>561</b>	2
manipulation, <b>519</b>	3
offset, <b>18</b> , <u>518</u>	4
pointer, $\underline{519}$	5
size, $\underline{519}$	6
view, 517, <u>518</u> , <b>530</b>	7
file size, <b>576</b>	
finished, <b>387</b>	8
Fortran	9
language binding, 20, 633	10
Fortran support, <b>633</b>	11
Freeing, $\underline{11}$	12
gather, <b>151</b>	13
nonblocking, 203	14
persistent, <b>220</b>	15
gather-to-all, 167	16
nonblocking, <b>207</b>	17
persistent, <b>224</b>	18
general datatype, $\underline{85}$	19
generalized requests, $501$ , $501$	20
get	20
in function names, $\underline{10}$	
graph	22
topology, $\underline{310}$ , $314$	23
group, <b>243</b> , <u>244</u> , <u>246</u> , 279 group objects, <u>246</u>	24
group objects, $\underline{240}$	25
handles, <u>14</u> , <b>682</b>	26
host rank, <b>361</b>	27
immediate 12 50	28
immediate, $\underline{12}$ , $\underline{50}$	29
inactive, <u>54</u> inclusive scan, <b>195</b>	30
nonblocking, <b>216</b>	31
persistent, 233	32
incomplete, $\underline{12}$ , $\underline{50}$	33
incomplete procedure, $\underline{12}$	34
independent, $\underline{423}$	35
individual file pointers, <u>533</u> , <b>540</b>	36
info object, <b>391</b>	37
file info, <b>526</b>	38
keys, <b>718</b>	39
values, <b>718</b>	
Initialization, $\underline{11}$	40
initiation, <u>11</u> , <b>50</b>	41
inter-communication, $\underline{245}$ , $\underline{278}$	42
inter-communicator, $\underline{245}$ , $\underline{278}$	43
collective operations, 147, 148	44
interlanguage communication, <b>694</b> internal	45
file data representation, $562$	46
interoperability, $561$	47
intra-communication, <u>245</u> , <u>278</u>	48
, <u></u>	

#### General Index

1 intra-communicator, 244, 278  $\mathbf{2}$ collective operations, 146 intra-communicator objects, 247 3 I/O, **517** 4 IO rank, **361** 5is 6 in function names, 10 7 language binding, 19, 633 8 interoperability, 681 9 summary, 697 10 lb_marker, 97, 98, 101, 107, 107, 111 11 erased, 110 12local, 3913 local group, 258 14local procedure, 1215loosely synchronous model, 306 lower bound, 10716lower-bound markers. 106 1718 macros, 21 19 main thread, 511 20matched receives. 72 21matching type, **35**, <u>114</u>, **575** 22 matching probe, 70 23memory 24allocation, 363 25system, 14 26memory model, 428, 461 27separate, 428, 434 28unified, 428, 434 message, 25 29 data, 27 30 envelope, 29  31 modes, 39 32 module variables, 674 33 mpi module 34 Fortran support, 637 35 MPI operation, 11 mpi_f08 module 36 Fortran support, 634 37 mpiexec, 382, 387, <u>388</u> 38 mpif.h include file 39 Fortran support, 639 40 mpirun, 388 41 multiple completions, 59 42named datatype, 13 43 names, **411** 44 name publishing, 416 45naming objects, 302 46 native 47file data representation, 562 48 neighborhood collective communication, 334

nonblocking, 344 non-local, 39, 40 non-local procedure,  $\underline{12}$ nonblocking, 49, 443, 533 communication, 49completion, 54 Fortran problems, 668 I/O, 534 initiation, 50 request objects, 50 nonblocking operation, 13nonblocking procedure, 12null handle, 54 null processes, 83 offset, 18, 518 one-sided communication, 427 Fortran problems, 669 opaque objects, 14, 686 operation is complete, 13 origin, <u>428</u> pack. 134 canonical, 140 packing unit, 136 parallel procedure, 307passive target communication, 462performance variables tools interface, 606 persistent communication requests, 57, 76 collective persistent, 217, 349 Fortran problems, 669 persistent operation, 13PMPI_, 587 point-to-point communication, 25 portable datatype, 14 ports, **411** predefined datatype, 13predefined reduction operations, 178 private window copy, 461 probe, 67 probe, matching, **70** process creation, 397 process group, 29 processes, 22 processor name, 362 profiling interface, 587 program error, 22 prototype definitions, 712 deprecated, 717 public window copy, 461

rank, <u>246</u> ready, <u>40</u>, 49, <u>50</u> nonblocking, <u>49</u>

ready send, 42receive, 25, 26, 30 buffer, 26complete,  $\underline{49}$ context, 279 start call,  $\underline{49}$ reduce, **176** nonblocking, 212 persistent, 229 reduce-scatter, 192 nonblocking, **214**, **215** persistent, 231, 232 reduction operations, 175, 689 predefined, 178 process-local, 191 scan, 195 user-defined, 185 related, 136 relative displacement, 18, 103 remote group, 258 Remote Memory Access, see RMA removed interfaces, 19, 629 request complete I/O, 534 request objects, 50 resource error,  $\underline{22}$ RMA, <u>427</u> communication calls, 443 request-based, 456 memory model, 461 synchronization calls, 462 scan, 195 inclusive, 195 scatter, 161 nonblocking, 205 persistent, 222 seek, **552** semantics, 835 file consistency, 570 nonblocking communications, 58 point-to-point communication, 43 semantics and correctness one-sided communication, 479 send, <u>25</u>, **26** buffer, 25 complete,  $\underline{49}$ context, 279 start,  $\underline{49}$ send-receive, 81 separate memory model, 428, 434, 461 sequential storage, 117Set in function names, 10

shared file pointers, <u>533</u> , <b>548</b>	1
shared memory allocation, 433	2
signals, 24	3
singleton init, <b>422</b>	
size changing	4
I/O, 576	5
	6
source, 279	7
split collective, 533, <b>554</b>	8
standard, $\underline{39}$ , 49	9
nonblocking, $50$	
standard send, 42	10
Starting, <u>11</u>	11
starting processes, <b>398</b> , <b>400</b>	12
startup, 381	13
portable, 388	14
state, <b>16</b>	15
status, <b>32</b> , <b>684</b>	
associating information, <b>508</b>	16
ignore, <b>34</b>	17
test, <b>66</b>	18
strong synchronization, $\underline{465}$	19
synchronization, $427$ , $443$	20
synchronization calls	21
RMA, <b>462</b>	
synchronizing, $\underline{13}$	22
synchronous, $40$ , $49$ , $50$ , $54$	23
nonblocking, $\underline{49}$	24
synchronous send, $42$	25
system memory, $\underline{14}$	26
	27
tag values, <b>361</b>	28
target, $\underline{428}$	
thread compliant, $510$ , $514$	29
threads, <b>510</b>	30
timers and synchronization, <b>380</b>	31
tool information interface, <b>593</b>	32
tool support, <b>587</b>	33
topologies, <b>309</b>	34
topology	
Cartesian, <u>310</u> , <b>312</b>	35
distributed graph, <u>310</u> , <b>316</b>	36
graph, $\underline{310}$ , $314$	37
virtual, <b>310</b>	38
true extent of datatypes, <b>110</b>	39
type map, <u>86</u>	40
type matching, $35$ , $114$	41
type signature, $\underline{86}$	
types, <b>710</b>	42
V <b>L</b> /	43
ub_marker, 97, 98, 101, 102, 107, <u>107</u> , 111	44
erased, $\underline{110}$	45
unified memory model, $428$ , $434$ , $461$	46
universe size, <b>421</b>	47
unnamed datatype, $\underline{13}$	
• • • —	48

1	unpack, <b>134</b>
2	canonical, 140
3	upper bound, $\underline{107}$
4	upper-bound markers, 106
5	user functions at process termination, 387
6	user-defined data representations, 565
7	user-defined reduction operations, 185
8	verbosity levels
9	tools interface, <b>594</b>
10	version inquiries, <b>359</b>
11	view, 517, <u>518</u> , <b>530</b>
12	virtual topology, $\underline{244}$ , $\underline{245}$ , $310$
13	weak synchronization, $465$
14	window
15	allocation, <b>431</b>
16	creation, 429
17	dynamically attached memory, <b>436</b> shared memory allocation, <b>433</b>
18	shared memory anocation, 455
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	

## **Examples Index**

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.

ASYNCHRONOUS, 495, 677, 679 Attributes between languages, 690 Basic tool using performance variables in the MPI tool information interface, 616 C/Fortran handle conversion, 683 Cartesian virtual topologies, 354 Client-server code, 65with blocking probe, 68with blocking probe, wrong, 69 Datatype 3D array, 125absolute addresses, 130 array of structures, 127 elaborate example, 137, 139 matching type, 114 matrix transpose, 126 union, 131Datatypes matching, 35 not matching, 36untyped, 36 Deadlock if not buffered, 45 with MPI_Bcast, 234, 235 wrong message exchange, 45False matching of collective operations, 238 Fortran 90 copying and sequence problem, 661, 663, 664 Fortran 90 derived types, 665 Fortran 90 heterogeneous communication, 658, 659 Fortran 90 invalid KIND, 654 Fortran 90 MPI_TYPE_MATCH_SIZE implementation, 657Fortran 90 overlapping communication and computation, 677, 678, 680 Fortran 90 register optimization, 668, 670

Independence of nonblocking operations, 241	13
Intercommunicator, 262, 266	14
Interlanguage communication, 694	15
Intertwined matching pairs, $43$	16
	17
Message exchange, 45	
Mixing blocking and nonblocking collective	18
operations, 237	19
Mixing collective and point-to-point requests,	20
240 MDL ACCUMULATE 451	21
MPI_ACCUMULATE, 451	22
MPI_Accumulate, 493, 496	23
MPI_Aint, 127	24
MPI_Aint_add, 496	25
MPI_Allgather, 169	
MPI_ALLOC_MEM, 365	26
MPI_Alloc_mem, 365, 496	27
MPI_ALLREDUCE, 190	28
MPI_Alltoall, 239	29
MPI_ASYNC_PROTECTS_NONBLOCKING,	30
495 MDI Domion 285 482 486 402 404	31
MPI_Barrier, 385, 483–486, 492–494	32
MPI_Bcast, 151, 234–238	33
MPI_BSEND, 43 MBI_Buffer_attach_47_284	
MPI_Buffer_attach, 47, 384 MPI_Buffer_detach, 47	34
MPI_Buner_detach, 47 MPI_BYTE, 36	35
MPI_Cancel, 385	36
	37
MPI_CART_COORDS, 331	38
MPI_CART_GET, 354 MPI_CART_RANK, 331	39
MPI_CART_RANK, 331 MPI_CART_SHIFT, 331, 354	40
MPI_CART_SUB, 332	41
MPI_CHARACTER, 37	
· · · · · · · · · · · · · · · · · · ·	42
MPI_Comm_create, 262, 272, 273, 276 MPI_Comm_create_keyval, 300	43
•	44
MPI_Comm_dup, 275 MPI_Comm_get_attr, 300	45
	46
MPI_Comm_group, 262, 276, 300	47
MPI_Comm_remote_size, 266	48
MPI_Comm_set_attr, 300	-40

10

- MPI_COMM_SPAWN_MULTIPLE, 408 3 MPI_Comm_spawn_multiple, 408 4 MPI_Comm_split, 266, 283, 285  $\mathbf{5}$
- MPI_Compare_and_swap, 494, 496 6
- MPI_DIMS_CREATE, 313, 354  $\overline{7}$
- MPI_DIST_GRAPH_CREATE, 320
- 8 MPI_Dist_graph_create, 321
- 9 MPI_DIST_GRAPH_CREATE_ADJACENT, 10 320
- MPI_F_sync_reg, 495 11
- MPI_FILE_CLOSE, 541, 544 12
- MPI_FILE_GET_AMODE, 525 13
- MPI_FILE_IREAD, 544 14
- MPI_FILE_OPEN, 541, 544 15
- MPI_FILE_READ, 541 16
- MPI_FILE_SET_ATOMICITY, 577
- 17MPI_FILE_SET_VIEW, 541, 544
- MPI_FILE_SYNC, 577 18 MPI_Finalize, 384, 385
- 19MPI_FREE_MEM, 365
- 20MPI_Free_mem, 496
- 21MPI_Gather, 139, 154, 155, 159
- 22MPI_Gatherv, 139, 156–159
- 23MPI_GET, 447, 448
- 24 MPI_Get, 483, 485, 491, 492
- 25MPI_Get_accumulate, 493, 496
- MPI_GET_ADDRESS, 104, 665, 666, 687 26
- MPI_Get_address, 127, 130, 131, 137 27
- MPI_GET_COUNT, 116 28
- MPI_GET_ELEMENTS, 116
- 29MPI_GRAPH_CREATE, 314, 327
- 30 MPI_GRAPH_NEIGHBORS, 327
- 31 MPI_GRAPH_NEIGHBORS_COUNT, 327
- 32MPI_Grequest_complete, 506
- $\rm MPI_Grequest_start, \ 506$ 33
- MPI_Group_excl, 272 34
- MPI_Group_free, 262, 272, 273 35
- MPI_Group_incl, 262, 273, 276 36
- MPI_Iallreduce, 240 37
- MPI_Ialltoall, 239 38
- MPI_Ibarrier, 237-240
- 39MPI_Ibcast, 202, 240, 241
- MPI_INFO_ENV, 383 40
- MPI_Intercomm_create, 283, 285 41
- MPI_Iprobe, 385 42
- MPI_IRECV, 56–58, 65 43
- MPI_Irecv, 240 44
- MPI_ISEND, 56-58, 65 45
- MPI_Op_create, 188, 189, 197
- 46MPI_Pack, 137, 139
- 47MPI_Pack_size, 139
- MPI_PROBE, 68, 69 48

MPI_Put, 468, 474, 484, 486, 490, 491 MPI_RECV, 35-37, 43, 45, 58, 68, 69, 114 MPI_Recv, 239 MPI_REDUCE, 179, 180, 183 MPI_Reduce, 183, 184, 188, 189 MPI_REQUEST_FREE, 57 MPI_Request_free, 384 MPI_Rget, 495 MPI_Rput, 495 MPI_Scan, 197 MPI_Scatter, 164 MPI_Scatterv, 164, 165 MPI_SEND, 35-37, 45, 58, 68, 69, 114 MPI_Send, 127, 130, 131, 137, 239, 240 MPI_SENDRECV, 125, 126 MPI_SENDRECV_REPLACE, 331 MPI_SSEND, 43, 58 MPI_Test_cancelled, 385 MPI_TYPE_COMMIT, 112, 125, 126, 447, 665,666 MPI_Type_commit, 127, 130, 131, 137, 155-159, 165, 197 MPI_TYPE_CONTIGUOUS, 88, 107, 114, 116 MPI_Type_contiguous, 155 MPI_TYPE_CREATE_DARRAY, 103 MPI_TYPE_CREATE_HVECTOR, 125, 126 MPI_Type_create_hvector, 127, 130 MPI_TYPE_CREATE_INDEXED_BLOCK, 447 MPI_TYPE_CREATE_RESIZED, 665, 666 MPI_TYPE_CREATE_STRUCT, 95, 107, 126, 665, 666 MPI_Type_create_struct, 127, 130, 131, 137, 158, 159, 197 MPI_TYPE_CREATE_SUBARRAY, 585 MPI_TYPE_EXTENT, 447

- MPI_TYPE_FREE, 447
- MPI_Type_get_contents, 132
- MPI_Type_get_envelope, 132
- MPI_TYPE_GET_EXTENT, 125, 126, 448, 451
- MPI_Type_get_extent, 127
- MPI_TYPE_INDEXED, 91, 125
- MPI_Type_indexed, 127, 130
- MPI_TYPE_VECTOR, 88, 89, 125, 126
- MPI_Type_vector, 156, 157, 159, 165
- MPI_Unpack, 137, 139
- MPI_User_function, 189
- MPI_WAIT, 56-58, 65, 544
- MPI_Wait, 237-240
- MPI_Waitall, 240, 495
- MPI_WAITANY, 65
- MPI_Waitany, 495 MPI_WAITSOME, 65

1

 $\mathbf{2}$ 

MPI_Win_attach, 496	1
MPI_Win_complete, 468, 485, 486, 491, 492	2
MPI_WIN_CREATE, 447, 448, 451	3
MPI_Win_create_dynamic, 496	4
MPI_Win_detach, 496	5
MPI_WIN_FENCE, 447, 448, 451	6
MPI_Win_fence, 490, 491	7
MPI_Win_flush, 484, 493, 494, 496	
MPI_Win_flush_all, 493	8
MPI_Win_flush_local, 483	9
MPI_WIN_FREE, 448, 451	10
MPI_Win_lock, 474, 483–486	11
MPI_Win_lock_all, 495, 496	12
MPI_Win_post, 485, 486, 491, 492	13
MPI_Win_start, 468, 485, 486, 491, 492	14
MPI_Win_sync, 483, 484, 493, 494	15
shared memory windows, 495	16
MPI_Win_unlock, 474, 483–486	
MPI_Win_unlock_all, 495, 496	17
MPI_Win_wait, 485, 486, 491, 492	18
mpiexec, 383, 389, 390	19
Neighborhood collective communication, 354	20
No Matching of Blocking and Nonblocking	21
collective operations, 239	22
Non-deterministic program with MPI_Bcast,	23
236	24
Non-overtaking messages, 43	25
Nonblocking operations, 56, 57	
message ordering, 58	26
progress, 58	27
	28
Overlapping Communicators, 240	29
	30
Pipelining nonblocking collective operations,	31
	32
Profiling interface, 590	33
Progression of nonblocking collective	34
operations, 239	35
Reading the value of a control variable in the	
MPI tool information interface, 605	36
	37
Shared memory windows	38
MPI_Win_sync, 495	39
	40
Threads and MPI, 511	41
Topologies, 354	42
Typemap, 87–89, 91, 95, 103	43
	40
Using MPI_T_CVAR_GET_INFO to list all	
names of control variables., $602$	45
Virtual topologies, 354	46
virtual topologics, 504	47
	48

# MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

5 6

12

13	MPI::_LONG_LONG, 840
14	MPI::BOOL, 839
15	MPI::COMPLEX, 839
16	MPI::DOUBLE_COMPLEX, 839
17	MPI::F_COMPLEX16, 840
18	MPI::F_COMPLEX32, 840
19	MPI::F_COMPLEX4, 840
20	MPI::F_COMPLEX8, 840
20	MPI::INTEGER16, 840
	MPI::LONG_DOUBLE_COMPLEX, 839
22	MPI::LONG_LONG, 840
23	MPI::REAL16, 840
24	MPI_2DOUBLE_PRECISION, 182, 703
25	MPI_2INT, 182, 703
26	MPI_2INTEGER, 182, 703
27	MPI_2REAL, 182, 703
28	MPI_ADDRESS_KIND, 17, 18, <u>18</u> , 28, 287,
29	660, 689, 700
	MPI_AINT, 27, 29, 179, 437, 701, 702, 845,
30	847
31	MPI_ANY_SOURCE, 30, 31, 43, 53, 54,
32	67-71, 79, 81, 82, 269, 308, 361, 699
33	MPI_ANY_TAG, 16, 17, 30, 31, 33, 53, 54,
34	67-71, 73, 74, 79, 81-83, 269, 699, 842
35	MPI_APPNUM, 422, 423, 706
36	MPI_ARGV_NULL, 17, 402, 403, 660, 708
37	MPI_ARGVS_NULL, 17, 407, 660, 708 MPI_ASYNC_PROTECTS_NONBLOCKING,
38	17, 495, 634, 635, 637, 639, 642, 649,
39	651, 671, 700, 844
	MPI_BAND, 178, 179, 704
40	MPI_BOR, 178, 179, 704
41	MPI_BOTTOM, 10, 17, 18, 34, 103, 117, 118,
42	147, 318, 320, 404, 437, 441, 636, 638,
43	645, 660, 665, 667, 669, 670, 672-674,
44	676, 687, 688, 694, 699, 851
45	MPI_BSEND_OVERHEAD, 49, 699
46	MPI_BXOR, 178, 179, 704
47	MPI_BYTE, 27, 28, 35, 36, 38, 140, 179, 518,
48	562, 575, 694, 701, 702, 848
-10	

MPI_C_BOOL, 28, 179, 701, 840, 845-847 MPI_C_COMPLEX, 28, 179, 701, 839, 845-847 MPI_C_DOUBLE_COMPLEX, 28, 179, 701, 845-847 MPI_C_FLOAT_COMPLEX, 179, 701, 845-847 MPI_C_LONG_DOUBLE_COMPLEX, 28, 179, 701, 845-847 MPI_CART, 322, 705 MPI_CHAR, 28, 38, 95, 180, 181, 597, 598, 701,845 MPI_CHARACTER, 27, 36-38, 180, 181, 702 MPI_COMBINER_CONTIGUOUS, 119, 122, 707 MPI_COMBINER_DARRAY, 119, 124, 707 MPI_COMBINER_DUP, 119, 122, 707 MPI_COMBINER_F90_COMPLEX, 119, 124, 707 MPI_COMBINER_F90_INTEGER, 119, 124, 707 MPI_COMBINER_F90_REAL, 119, 124, 707 MPI_COMBINER_HINDEXED, 20, 119, 123, 707 MPI_COMBINER_HINDEXED_BLOCK, 119, 123, 707, 841 MPI_COMBINER_HINDEXED_INTEGER, 20, 630, 840 MPI_COMBINER_HVECTOR, 20, 119, 123, 707 MPI_COMBINER_HVECTOR_INTEGER, 20, 630, 840 MPI_COMBINER_INDEXED, 119, 123, 707 MPI_COMBINER_INDEXED_BLOCK, 119, 123, 707 MPI_COMBINER_NAMED, 119, 122, 707 MPI_COMBINER_RESIZED, 119, 125, 707 MPI_COMBINER_STRUCT, 20, 119, 123, 707

MPI_COMBINER_STRUCT_INTEGER, 20, 630, 840

MPI_COMBINER_SUBARRAY, 119, 124, 707
MPI_COMBINER_VECTOR, 119, 122, 707
MPI_COMM_DUP_FN, 20, <u>290</u> , 705, 844
MPI_COMM_NULL, 247, 261, 262, 264–266,
268, 304, 312, 314, 405, 424–426, 704,
848
MPI_COMM_NULL_COPY_FN, 20, 290, 636,
688, 705, 844
MPI_COMM_NULL_DELETE_FN, 20, <u>290</u> ,
MPI_COMM_PARENT, 304
MPI_COMM_SELF, 23, 247, 264, 270, 287,
304, 366, 367, 387, 424, 520, 631, 703,
836, 846
MPI_COMM_TYPE_SHARED, 268, 703, 842
MPI_COMM_WORLD, 17, 24, 29, 30,
247-249, 256, 257, 270, 273, 282, 304,
313, 360-362, 367, 369, 377, 384-388,
397, 398, 400, 401, 405, 407, 421-424,
514, 561, 581, 604, 612, 631, 682, 693,
703, 836, 850
MPI_COMPLEX, 27, 179, 564, 652, 702
MPI_COMPLEX16, 179, 702
MPI_COMPLEX32, 179, 702
MPI_COMPLEX4, 179, 702
MPI_COMPLEX8, 179, 702
MPI_CONGRUENT, 257, 280, 703
MPI_CONVERSION_FN_NULL, <u>569</u> , 705
MPI_COUNT, 27, 29, 179, 597, 701, 702, 841
MPI_COUNT_KIND, 17, 28, 700
MPI_CXX_BOOL, 29, 179, 702, 839
MPI_CXX_DOUBLE_COMPLEX, 29, 179,
702, 839
MPI_CXX_FLOAT_COMPLEX, 29, 179, 702,
839
MPI_CXX_LONG_DOUBLE_COMPLEX, 29,
179, 702, 839
MPI_DATATYPE_NULL, 113, 704
MPI_DISPLACEMENT_CURRENT, 530,
531, 708, 851
MPI_DIST_GRAPH, 322, 705, 846
MPI_DISTRIBUTE_BLOCK, 100, 708
MPI_DISTRIBUTE_CYCLIC, 100, 708
MPI_DISTRIBUTE_DFLT_DARG, 100, 708
MPI_DISTRIBUTE_NONE, 100, 708
MPI_DOUBLE, 28, 178, 597, 606–608, 651,
701
MPI_DOUBLE_COMPLEX, 27, 179, 564, 652,
702
MPI_DOUBLE_INT, 182, 183, 703
MPI_DOUBLE_PRECISION, 27, 178, 652,
702
MPI_DUP_FN, 20, 290, <u>626</u> , 706
MPI_ERR_ACCESS, 376, 523, 582, 698

MDI EDD AMODE 276 FOI FOO 600	1
MPI_ERR_AMODE, 376, 521, 582, 698	
MPI_ERR_ARG, 375, 697	2
MPI_ERR_ASSERT, 375, 478, 698	3
MPI_ERR_BAD_FILE, 376, 582, 698	4
MPI_ERR_BASE, 364, 375, 478, 698	5
MPI_ERR_BUFFER, 375, 697	6
MPI_ERR_COMM, 375, 697	
MPI_ERR_CONVERSION, 376, 570, 582, 698	7
MPI_ERR_COUNT, 375, 697	8
MPI_ERR_DIMS, 375, 697	9
MPI_ERR_DISP, 375, 478, 698	10
MPI_ERR_DUP_DATAREP, 376, 567, 582,	11
698	12
MPI_ERR_FILE, 376, 582, 698	
MPI_ERR_FILE_EXISTS, 376, 582, 698	13
MPI_ERR_FILE_IN_USE, 376, 523, 582, 698	14
MPI_ERR_GROUP, 375, 697	15
MPI_ERR_IN_STATUS, 32, 34, 55, 61, 63,	16
368, 375, 505, 535, 698	17
MPI_ERR_INFO, 375, 698	
MPI_ERR_INFO_KEY, 375, 392, 698	18
MPI_ERR_INFO_NOKEY, 375, 393, 698	19
MPI_ERR_INFO_VALUE, 375, 392, 698	20
MPI_ERR_INTERN, 367, 375, 697	21
MPI_ERR_IO, 376, 582, 698	22
MPI_ERR_KEYVAL, 300, 375, 698	23
$MII 1_DIUU_IUDI VAL, 500, 575, 050$	
MPLERR LASTCODE 374 376 377 370	24
MPI_ERR_LASTCODE, 374, 376, 377, 379,	24 25
622, 699	25
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698	
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698	25
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698	25 26
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698	25 26 27
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582,	25 26 27 28
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698	25 26 27 28 29 30
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_NOT_SAME, 376, 582, 698	25 26 27 28 29 30 31
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_NOT_SAME, 376, 582, 698 MPI_ERR_OP, 375, 478, 697	25 26 27 28 29 30
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_NOT_SAME, 376, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697	25 26 27 28 29 30 31
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_NOT_SAME, 376, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697	25 26 27 28 29 30 31 32
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_NOT_SAME, 376, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698	25 26 27 28 29 30 31 32 33
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_NOT_SAME, 376, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_QUOTA, 376, 582, 698	25 26 27 28 29 30 31 32 33 34 35
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_QUOTA, 376, 582, 698 MPI_ERR_RANK, 375, 478, 697	25 26 27 28 29 30 31 32 33 34 35 36
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 376, 582, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697	25 26 27 28 29 30 31 32 33 34 35 36 37
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_REQUEST, 375, 697	25 26 27 28 29 30 31 32 33 34 35 36 37 38
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_QUOTA, 376, 582, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698	25 26 27 28 29 30 31 32 33 34 35 36 37
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_QUOTA, 376, 582, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_RAMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_CONFLICT, 375, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_RAMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_CONFLICT, 375, 478, 698 MPI_ERR_RMA_FLAVOR, 376, 435, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_RAMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_FLAVOR, 376, 435, 478, 698 MPI_ERR_RMA_RANGE, 376, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OTHER, 374, 375, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 376, 582, 698 MPI_ERR_RANK, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 478, 698 MPI_ERR_RMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_FLAVOR, 376, 435, 478, 698 MPI_ERR_RMA_RANGE, 376, 478, 698 MPI_ERR_RMA_RANGE, 376, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 376, 582, 698 MPI_ERR_RANK, 375, 415, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 478, 698 MPI_ERR_RMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_FLAVOR, 376, 435, 478, 698 MPI_ERR_RMA_RANGE, 376, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SYNC, 375, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 478, 698 MPI_ERR_RMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SYNC, 375, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_RANK, 375, 697 MPI_ERR_RAMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_ATTACH, 376, 435, 478, 698 MPI_ERR_RMA_FLAVOR, 376, 435, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SYNC, 375, 478, 698 MPI_ERR_RMA_SYNC, 375, 478, 698 MPI_ERR_RMA_SENVICE, 375, 417, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_RAMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_ATTACH, 376, 435, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SYNC, 375, 417, 698 MPI_ERR_SIZE, 375, 478, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
622, 699 MPI_ERR_LOCKTYPE, 375, 478, 698 MPI_ERR_NAME, 375, 418, 698 MPI_ERR_NO_MEM, 364, 375, 698 MPI_ERR_NO_SPACE, 376, 582, 698 MPI_ERR_NO_SUCH_FILE, 376, 522, 582, 698 MPI_ERR_OP, 375, 478, 697 MPI_ERR_OP, 375, 478, 697 MPI_ERR_PENDING, 61, 375, 697 MPI_ERR_PORT, 375, 415, 698 MPI_ERR_RANK, 375, 478, 698 MPI_ERR_RANK, 375, 478, 697 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_READ_ONLY, 376, 582, 698 MPI_ERR_RANK, 375, 697 MPI_ERR_RAMA_ATTACH, 376, 478, 698 MPI_ERR_RMA_ATTACH, 376, 435, 478, 698 MPI_ERR_RMA_FLAVOR, 376, 435, 478, 698 MPI_ERR_RMA_SHARED, 376, 478, 698 MPI_ERR_RMA_SYNC, 375, 478, 698 MPI_ERR_RMA_SYNC, 375, 478, 698 MPI_ERR_RMA_SENVICE, 375, 417, 698	25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

1 MPI_ERR_TOPOLOGY, 375, 697  $\mathbf{2}$ MPI_ERR_TRUNCATE, 375, 697 MPI_ERR_TYPE, 375, 697 3 MPI_ERR_UNKNOWN, 374, 375, 697 4 MPI_ERR_UNSUPPORTED_DATAREP, 376,  $\mathbf{5}$ 582.698 6 MPI_ERR_UNSUPPORTED_OPERATION, 7 376, 582, 698 8 MPI_ERR_WIN, 375, 478, 698 9 MPI_ERRCODES_IGNORE, 17, 404, 660, 708 10 MPI_ERRHANDLER_NULL, 373, 704 MPI_ERROR, 32, 55, 199, 456, 700, 837, 843 11 MPI_ERRORS_ABORT, 366, 836 12MPI_ERRORS_ARE_FATAL, 366, 367, 379, 13478, 581, 700, 836 14MPI_ERRORS_RETURN, 366, 367, 380, 386, 15581, 693, 700 16MPI_F08_STATUS_IGNORE, 685, 709, 843 17MPI_F08_STATUSES_IGNORE, 685, 709, 843 MPI_F_STATUS_IGNORE, 684, 709 18 MPI_F_STATUSES_IGNORE, 684, 709 19MPI_FILE_NULL, 522, 581, 704 20MPI_FLOAT, 28, 95, 176, 178, 563, 701 21MPI_FLOAT_INT, 13, 182, 183, 703 22MPI_GRAPH, 322, 705 23MPI_GROUP_EMPTY, 246, 252, 261, 262,  24 264.704 25MPI_GROUP_NULL, 246, 255, 704 MPI_HOST, 361, 703 26MPI_IDENT, 249, 257, 703 27MPI_IN_PLACE, 17, 146, 173, 639, 660, 699 28MPI_INFO, 219 29MPI_INFO_ENV, 382, 383, 703, 842 30 MPI_INFO_NULL, 320, 396, 404, 413, 521,  31 522, 532, 704 MPI_INT, 13, 28, 86, 178, 563, 564, 597, 598, 32 601, 606, 610, 651, 693, 695, 701 33 MPI_INT16_T, 28, 178, 701, 845, 847 34MPI_INT32_T, 28, 178, 701, 845, 847 35 MPI_INT64_T, 28, 178, 701, 845, 847 36 MPI_INT8_T, 28, 178, 701, 845, 847 37 MPI_INTEGER, 27, 35, 178, 651, 652, 695, 38 702 39 MPI_INTEGER1, 27, 178, 702 MPI_INTEGER16, 178, 702 40MPI_INTEGER2, 27, 178, 564, 702 41 MPI_INTEGER4, 27, 178, 702 42MPI_INTEGER8, 178, 655, 702 43MPI_INTEGER_KIND, 17, 700 44MPI_IO, 361, 703 45MPI_KEYVAL_INVALID, 290-292, 699 46MPI_LAND, 178, 179, 704 47MPI_LASTUSEDCODE, 377, 378, 706 MPI_LB, 20, 630, 840 48

MPI_LOCK_EXCLUSIVE, 471, 699 MPI_LOCK_SHARED, 471, 472, 699 MPI_LOGICAL, 27, 179, 702 MPI_LONG, 28, 178, 701 MPI_LONG_DOUBLE, 28, 178, 701 MPI_LONG_DOUBLE_INT, 182, 703 MPI_LONG_INT, 182, 183, 703 MPI_LONG_LONG, 28, 178, 701, 847 MPI_LONG_LONG_INT, 28, 178, 701, 847 MPI_LOR, 178, 179, 704 MPI_LXOR, 178, 179, 704 MPI_MAX, 176, 178, 179, 197, 704 MPI_MAX_DATAREP_STRING, 17, 532, 567,700 MPI_MAX_ERROR_STRING, 17, 373, 378, 700 MPI_MAX_INFO_KEY, 17, 375, 391, 394, 700 MPI_MAX_INFO_VAL, 17, 375, 391, 700 MPI_MAX_LIBRARY_VERSION_STRING, 17, 360, 700, 840 MPI_MAX_OBJECT_NAME, 17, 303-305, 700, 842, 848 MPI_MAX_PORT_NAME, 17, 413, 700 MPI_MAX_PROCESSOR_NAME, 17, 362, 363, 700, 849 MPI_MAXLOC, 178, 181, 182, 185, 704 MPI_MESSAGE_NO_PROC, 71, 73, 74, 699, 841 MPI_MESSAGE_NULL, 71, 73, 74, 704, 841 MPI_MIN, 178, 179, 704 MPI_MINLOC, 178, 181, 182, 185, 704 MPI_MODE_APPEND, 520, 521, 707 MPI_MODE_CREATE, 520, 521, 529, 707 MPI_MODE_DELETE_ON_CLOSE, 520-522, 707 MPI_MODE_EXCL, 520, 521, 707 MPI_MODE_NOCHECK, 472, 476, 477, 707 MPI_MODE_NOPRECEDE, 467, 476, 477, 707MPI_MODE_NOPUT, 476, 477, 707 MPI_MODE_NOSTORE, 476, 477, 707 MPI_MODE_NOSUCCEED, 476, 477, 707 MPI_MODE_RDONLY, 520, 521, 526, 707 MPI_MODE_RDWR, 520, 521, 707 MPI_MODE_SEQUENTIAL, 520, 521, 523, 524, 530, 535, 541, 552, 573, 707, 851 MPI_MODE_UNIQUE_OPEN, 520, 521, 707 MPI_MODE_WRONLY, 520, 521, 707 MPI_NO_OP, 430, 453, 454, 704, 837 MPI_NULL_COPY_FN, 20, 290, 626, 706 MPI_NULL_DELETE_FN, 20, 290, 626, 706 MPI_OFFSET, 27, 179, 701, 702, 845, 847 MPI_OFFSET_KIND, 17, 18, 28, 575, 660, 700 MPI_OP_NULL, 188, 704

MPI_ORDER_C, 16, 97, 100, 101, 708
MPI_ORDER_FORTRAN, 16, 97, 100, 708
MPI_PACKED, 13, 27, 28, 35, 134, 136, 140,
565, 694, 701, 702
MPI_PROC_NULL, 26, 29, 31, 32, 68, 71, 73,
74, 83, 148, 150, 152, 154, 162, 164,
177, 249, 330, 334, 361, 435, 444, 699,
841, 848, 850
MPI_PROD, 178, 179, 704
MPI_REAL, 27, 35, 178, 564, 651, 652, 658,
702
MPI_REAL16, 179, 702
MPI_REAL2, 27, 179, 702
MPI_REAL4, 27, 179, 651, 655, 702
MPI_REAL8, 27, 179, 651, 702, 845
MPI_REPLACE, 451–454, 494, 704, 847, 850
MPI_REQUEST_NULL, 54–57, 60–63, 504,
704
MPI_ROOT, 148, 699
MPI_SEEK_CUR, 547, 553, 708
MPI_SEEK_END, 547, 553, 708
MPI_SEEK_SET, 547, 553, 708
MPI_SHORT, 28, 178, 701
MPI_SHORT_INT, 182, 703
MPI_SIGNED_CHAR, 28, 178, 180, 181, 701,
847 NDL CINILAD 240 255 200 500
MPI_SIMILAR, 249, 257, 280, 703
MPI_SOURCE, 32, 200, 700, 837, 843
MPI_STATUS_IGNORE, 10, 17, 34, 503, 535,
638,660,684,685,694,708,709,841
MPI_STATUS_SIZE, 17, 32, 640, 700, 843
MPI_STATUSES_IGNORE, 16, 17, 34, 503,
505, 660, 684, 685, 708, 709
MPI_SUBARRAYS_SUPPORTED, 17, 634,
635, 638-642, 646-649, 661-663, 700,
843
MPI_SUBVERSION, 17, 360, 709
MPI_SUCCESS, 21, 54, 61, 63, 289, 290,
292–295, 297, 298, 313, 374, 375, 379,
380, 404, 570, 593, 602, 610, 613, 615,
620,  622,  624,  626,  697,  836
MPI_SUM, 178, 179, 450, 693, 704
MPI_T_BIND_MPI_COMM, 595, 710
MPI_T_BIND_MPI_DATATYPE, 595, 710
MPI_T_BIND_MPI_ERRHANDLER, 595,
710
MPI_T_BIND_MPI_FILE, 595, 710
MPI_T_BIND_MPI_GROUP, 595, 710
MPI_T_BIND_MPI_INFO, 595, 710
MPI_T_BIND_MPI_MESSAGE, 595, 710
MPI_T_BIND_MPI_OP, 595, 710
MPI_T_BIND_MPI_REQUEST, 595, 710
MPI_T_BIND_MPI_WIN, 595, 710

MPI_T_BIND_NO_OBJECT, 595, 601, 603,	1
610,612,710	2
MPI_T_CVAR_HANDLE_NULL, 604, 709	3
MPI_T_CVAR_READ, WRITE, 624	4
MPI_T_ENUM_NULL, 601, 610, 709	5
MPI_T_ERR_XXX, 622	6
MPI_T_ERR_CANNOT_INIT, 624, 699	7
MPI_T_ERR_CVAR_SET_NEVER, 605, 624,	8
699	
MPI_T_ERR_CVAR_SET_NOT_NOW, 605,	9
624, 699	10
MPI_T_ERR_INVALID, 624, 699, 839	11
MPI_T_ERR_INVALID_HANDLE, 612, 624, 699	12
MPI_T_ERR_INVALID_INDEX, 20, 624, 628,	13
699, 836	14
MPI_T_ERR_INVALID_ITEM, 20, 624, 628,	15
699, 836	16
MPI_T_ERR_INVALID_NAME, 602, 610,	17
620, 624, 699, 839	18
MPI_T_ERR_INVALID_SESSION, 624, 699	
MPI_T_ERR_MEMORY, 624, 699	19
MPI_T_ERR_NOT_INITIALIZED, 624, 699	20
MPI_T_ERR_OUT_OF_HANDLES, 624, 699	21
MPI_T_ERR_OUT_OF_SESSIONS, 624, 699	22
MPI_T_ERR_PVAR_NO_ATOMIC, 615, 624,	23
699	24
MPI_T_ERR_PVAR_NO_STARTSTOP, 613,	25
624, 699	26
MPI_T_ERR_PVAR_NO_WRITE, 614, 615,	27
624,699	28
MPI_T_PVAR_ALL_HANDLES, 613–616, 710	29
MPI_T_PVAR_CLASS_AGGREGATE, 607,	
608, 710	30
MPI_T_PVAR_CLASS_COUNTER, 607, 710	31
MPI_T_PVAR_CLASS_GENERIC, 608, 710	32
MPI_T_PVAR_CLASS_HIGHWATERMARK,	33
607, 710	34
MPI_T_PVAR_CLASS_LEVEL, 606, 710	35
MPI_T_PVAR_CLASS_LOWWATERMARK,	36
607,710	37
MPI_T_PVAR_CLASS_PERCENTAGE, 607,	38
710 MPI_T_PVAR_CLASS_SIZE, 606, 710	39
MPI_T_PVAR_CLASS_STATE, 606, 710	40
MPI_T_PVAR_CLASS_STATE, 608, 710	
MPI_T_PVAR_HANDLE_NULL, 613, 709	41
MPI_T_PVAR_SESSION_NULL, 611, 709	42
MPI_T_SCOPE_ALL, 601, 710	43
MPI_T_SCOPE_ALL_EQ, 601, 605, 710	44
MPI_T_SCOPE_CONSTANT, 601, 710	45
MPI_T_SCOPE_GROUP, 601, 710	46
MPI_T_SCOPE_GROUP_EQ, 601, 605, 710	47
MPI_T_SCOPE_LOCAL, 601, 710	48

1	
	MPI_T_SCOPE_READONLY, 601, 710
2	MPI_T_VERBOSITY_MPIDEV_ALL, 594,
3	709
	MPI_T_VERBOSITY_MPIDEV_BASIC, 594,
4	
5	
6	MPI_T_VERBOSITY_MPIDEV_DETAIL,
	594,709
7	MPI_T_VERBOSITY_TUNER_ALL, 594, 709
8	MPI_T_VERBOSITY_TUNER_BASIC, 594,
9	709
10	MPI_T_VERBOSITY_TUNER_DETAIL, 594,
11	709
	MPI_T_VERBOSITY_USER_ALL, 594, 709
12	MPI_T_VERBOSITY_USER_BASIC, 594,
13	709
14	MPI_T_VERBOSITY_USER_DETAIL, 594,
15	709
16	
	MPI_TAG, 32, 200, 700, 837, 843
17	MPI_TAG_UB, 29, 361, 689, 692, 703
18	MPI_THREAD_FUNNELED, 514, 707
19	MPI_THREAD_MULTIPLE, 514, 516, 707,
20	838
21	MPI_THREAD_SERIALIZED, 514, 707
	MPI_THREAD_SINGLE, 514, 515, 707
22	MPI_TYPE_DUP_FN, <u>297</u> , 705
23	MPI_TYPE_NULL_COPY_FN, <u>297</u> , 705
24	MPI_TYPE_NULL_DELETE_FN, <u>297</u> , 705,
25	844
26	MPI_TYPECLASS_COMPLEX, 657, 708
	MPI_TYPECLASS_INTEGER, 657, 708
27	MPI_TYPECLASS_REAL, 657, 708
28	
29	MPI_UB, 4, 20, 630, 840 MPI_UINT16_T_28_178_701_845_847
29 30	MPI_UINT16_T, 28, 178, 701, 845, 847
30	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847
30 31	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847
30	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847
30 31	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106,
30 31 32	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266,
30 31 32 33 34	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847
30 31 32 33 34 35	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266,
30 31 32 33 34 35 36	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847
30 31 32 33 34 35	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847 MPI_UNEQUAL, 249, 257, 280, 703
30 31 32 33 34 35 36	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847 MPI_UNEQUAL, 249, 257, 280, 703 MPI_UNIVERSE_SIZE, 400, 421, 422, 706
30 31 32 33 34 35 36 37	<ul> <li>MPI_UINT16_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT32_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT64_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT8_T, 28, 178, 701, 845, 847</li> <li>MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847</li> <li>MPI_UNEQUAL, 249, 257, 280, 703</li> <li>MPI_UNIVERSE_SIZE, 400, 421, 422, 706</li> <li>MPI_UNSIGNED, 28, 178, 597, 606–608, 701</li> </ul>
30 31 32 33 34 35 36 37 38	MPI_UINT16_T, 28, 178, 701, 845, 847 MPI_UINT32_T, 28, 178, 701, 845, 847 MPI_UINT64_T, 28, 178, 701, 845, 847 MPI_UINT8_T, 28, 178, 701, 845, 847 MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847 MPI_UNEQUAL, 249, 257, 280, 703 MPI_UNIVERSE_SIZE, 400, 421, 422, 706 MPI_UNSIGNED, 28, 178, 597, 606–608, 701 MPI_UNSIGNED_CHAR, 28, 178, 180, 181, 701
30 31 32 33 34 35 36 37 38 39 40	<ul> <li>MPI_UINT16_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT32_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT64_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT8_T, 28, 178, 701, 845, 847</li> <li>MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847</li> <li>MPI_UNEQUAL, 249, 257, 280, 703</li> <li>MPI_UNIVERSE_SIZE, 400, 421, 422, 706</li> <li>MPI_UNSIGNED, 28, 178, 597, 606–608, 701</li> <li>MPI_UNSIGNED_LONG, 28, 178, 597,</li> </ul>
30 31 32 33 34 35 36 37 38 39 40 41	<ul> <li>MPI_UINT16_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT32_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT64_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT8_T, 28, 178, 701, 845, 847</li> <li>MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847</li> <li>MPI_UNEQUAL, 249, 257, 280, 703</li> <li>MPI_UNIVERSE_SIZE, 400, 421, 422, 706</li> <li>MPI_UNSIGNED, 28, 178, 597, 606–608, 701</li> <li>MPI_UNSIGNED_LONG, 28, 178, 597, 606–608, 701</li> </ul>
30 31 32 33 34 35 36 37 38 39 40 41 42	$\begin{array}{l} {\rm MPI_UINT16_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT32_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT32_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT64_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT8_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UNDEFINED,\ 33,\ 60,\ 61,\ 63,\ 64,\ 106, \\ 109,\ 111,\ 116,\ 137,\ 248,\ 249,\ 265,\ 266, \\ 322,\ 332,\ 333,\ 653,\ 699,\ 841,\ 847 \\ \\ {\rm MPI_UNEQUAL,\ 249,\ 257,\ 280,\ 703} \\ {\rm MPI_UNVERSE_SIZE,\ 400,\ 421,\ 422,\ 706} \\ {\rm MPI_UNSIGNED,\ 28,\ 178,\ 597,\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG,\ 28,\ 178,\ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \\ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \\ 597, \\ 606-608,\ 701} \\ \end{array}$
30 31 32 33 34 35 36 37 38 39 40 41	<ul> <li>MPI_UINT16_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT32_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT64_T, 28, 178, 701, 845, 847</li> <li>MPI_UINT8_T, 28, 178, 701, 845, 847</li> <li>MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, 109, 111, 116, 137, 248, 249, 265, 266, 322, 332, 333, 653, 699, 841, 847</li> <li>MPI_UNEQUAL, 249, 257, 280, 703</li> <li>MPI_UNVERSE_SIZE, 400, 421, 422, 706</li> <li>MPI_UNSIGNED, 28, 178, 597, 606–608, 701</li> <li>MPI_UNSIGNED_LONG, 28, 178, 597, 606–608, 701</li> <li>MPI_UNSIGNED_LONG_LONG, 28, 178, 597, 606–608, 701, 847</li> </ul>
30 31 32 33 34 35 36 37 38 39 40 41 42	$\begin{array}{l} {\rm MPI_UINT16_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT32_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT32_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT64_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT8_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UNDEFINED,\ 33,\ 60,\ 61,\ 63,\ 64,\ 106, \\ 109,\ 111,\ 116,\ 137,\ 248,\ 249,\ 265,\ 266, \\ 322,\ 332,\ 333,\ 653,\ 699,\ 841,\ 847 \\ \\ {\rm MPI_UNEQUAL,\ 249,\ 257,\ 280,\ 703} \\ {\rm MPI_UNVERSE_SIZE,\ 400,\ 421,\ 422,\ 706} \\ {\rm MPI_UNSIGNED,\ 28,\ 178,\ 597,\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG,\ 28,\ 178,\ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \\ 597,\ 606-608,\ 701,\ 847} \\ \\ {\rm MPI_UNSIGNED_SHORT,\ 28,\ 178,\ 701} \\ \end{array}$
30 31 32 33 34 35 36 37 38 39 40 41 42 43	$\begin{split} & \text{MPI_UINT16_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT32_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT64_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT64_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT8_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, \\ & 109, 111, 116, 137, 248, 249, 265, 266, \\ & 322, 332, 333, 653, 699, 841, 847 \\ & \text{MPI_UNEQUAL, 249, 257, 280, 703} \\ & \text{MPI_UNEQUAL, 249, 257, 280, 703} \\ & \text{MPI_UNIVERSE_SIZE, 400, 421, 422, 706} \\ & \text{MPI_UNSIGNED, 28, 178, 597, 606-608, 701} \\ & \text{MPI_UNSIGNED_LONG, 28, 178, 180, 181, \\ & 701 \\ & \text{MPI_UNSIGNED_LONG, 28, 178, 597, \\ & 606-608, 701 \\ & \text{MPI_UNSIGNED_LONG_LONG, 28, 178, \\ & 597, 606-608, 701, 847 \\ & \text{MPI_UNSIGNED_SHORT, 28, 178, 701} \\ & \text{MPI_UNWEIGHTED, 17, 317, 318, 320, 321, } \end{split}$
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	$\begin{split} & \text{MPI_UINT16_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT32_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT64_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT64_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT8_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, \\ & 109, 111, 116, 137, 248, 249, 265, 266, \\ & 322, 332, 333, 653, 699, 841, 847 \\ & \text{MPI_UNEQUAL, 249, 257, 280, 703} \\ & \text{MPI_UNEQUAL, 249, 257, 280, 703} \\ & \text{MPI_UNSIGNED, 28, 178, 597, 606-608, 701} \\ & \text{MPI_UNSIGNED_LONG, 28, 178, 180, 181, \\ & 701 \\ & \text{MPI_UNSIGNED_LONG, 28, 178, 597, \\ & 606-608, 701 \\ & \text{MPI_UNSIGNED_LONG_LONG, 28, 178, \\ & 597, 606-608, 701, 847 \\ & \text{MPI_UNSIGNED_SHORT, 28, 178, 701} \\ & \text{MPI_UNWEIGHTED, 17, 317, 318, 320, 321, \\ & 328, 329, 660, 708, 840, 846 \\ \end{split}$
<ol> <li>30</li> <li>31</li> <li>32</li> <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> </ol>	$\begin{array}{l} {\rm MPI_UINT16_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT32_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT64_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT8_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT8_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UNDEFINED,\ 33,\ 60,\ 61,\ 63,\ 64,\ 106, \\ 109,\ 111,\ 116,\ 137,\ 248,\ 249,\ 265,\ 266, \\ 322,\ 332,\ 333,\ 653,\ 699,\ 841,\ 847 \\ \\ {\rm MPI_UNEQUAL,\ 249,\ 257,\ 280,\ 703} \\ {\rm MPI_UNVERSE_SIZE,\ 400,\ 421,\ 422,\ 706} \\ {\rm MPI_UNSIGNED,\ 28,\ 178,\ 597,\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG,\ 28,\ 178,\ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_SHORT,\ 28,\ 178,\ 701} \\ \\ {\rm MPI_UNWEIGHTED,\ 17,\ 317,\ 318,\ 320,\ 321, \ 328,\ 329,\ 660,\ 708,\ 840,\ 846} \\ \\ {\rm MPI_VAL,\ 14,\ 682} \\ \end{array}$
<ol> <li>30</li> <li>31</li> <li>32</li> <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> </ol>	$\begin{split} & \text{MPI_UINT16_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT32_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT64_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UINT8_T, 28, 178, 701, 845, 847} \\ & \text{MPI_UNDEFINED, 33, 60, 61, 63, 64, 106, } \\ & 109, 111, 116, 137, 248, 249, 265, 266, \\ & 322, 332, 333, 653, 699, 841, 847 \\ & \text{MPI_UNEQUAL, 249, 257, 280, 703} \\ & \text{MPI_UNIVERSE_SIZE, 400, 421, 422, 706} \\ & \text{MPI_UNIVERSE_SIZE, 400, 421, 422, 706} \\ & \text{MPI_UNSIGNED_CHAR, 28, 178, 180, 181, } \\ & 701 \\ & \text{MPI_UNSIGNED_LONG, 28, 178, 597, } \\ & 606-608, 701 \\ & \text{MPI_UNSIGNED_LONG_LONG, 28, 178, } \\ & 597, 606-608, 701, 847 \\ & \text{MPI_UNSIGNED_SHORT, 28, 178, 701} \\ & \text{MPI_UNSIGNED_SHORT, 28, 178, 701} \\ & \text{MPI_UNSIGNED_SHORT, 28, 178, 320, 321, } \\ & 328, 329, 660, 708, 840, 846 \\ & \text{MPI_VAL, 14, 682} \\ & \text{MPI_VERSION, 17, 360, 709} \\ \end{split}$
<ol> <li>30</li> <li>31</li> <li>32</li> <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> </ol>	$\begin{array}{l} {\rm MPI_UINT16_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT32_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT64_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT8_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UINT8_T,\ 28,\ 178,\ 701,\ 845,\ 847} \\ {\rm MPI_UNDEFINED,\ 33,\ 60,\ 61,\ 63,\ 64,\ 106, \\ 109,\ 111,\ 116,\ 137,\ 248,\ 249,\ 265,\ 266, \\ 322,\ 332,\ 333,\ 653,\ 699,\ 841,\ 847 \\ \\ {\rm MPI_UNEQUAL,\ 249,\ 257,\ 280,\ 703} \\ {\rm MPI_UNVERSE_SIZE,\ 400,\ 421,\ 422,\ 706} \\ {\rm MPI_UNSIGNED,\ 28,\ 178,\ 597,\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG,\ 28,\ 178,\ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_LONG_LONG,\ 28,\ 178, \ 597, \\ 606-608,\ 701} \\ \\ {\rm MPI_UNSIGNED_SHORT,\ 28,\ 178,\ 701} \\ \\ {\rm MPI_UNWEIGHTED,\ 17,\ 317,\ 318,\ 320,\ 321, \ 328,\ 329,\ 660,\ 708,\ 840,\ 846} \\ \\ {\rm MPI_VAL,\ 14,\ 682} \\ \end{array}$

MPI_WEIGHTS_EMPTY, 17, 318, 320, 660, 708, 840 MPI_WIN_BASE, 440, 693, 706 MPI_WIN_CREATE_FLAVOR, 440, 706 MPI_WIN_DISP_UNIT, 440, 706 MPI_WIN_DUP_FN, 294, 705 MPI_WIN_FLAVOR_ALLOCATE, 441, 706 MPI_WIN_FLAVOR_CREATE, 441, 706 MPI_WIN_FLAVOR_DYNAMIC, 441, 706 MPI_WIN_FLAVOR_SHARED, 435, 441, 706 MPI_WIN_MODEL, 440, 462, 706 MPI_WIN_NULL, 439, 704 MPI_WIN_NULL_COPY_FN, 294, 705 MPI_WIN_NULL_DELETE_FN, 294, 705 MPI_WIN_SEPARATE, 441, 462, 480, 706 MPI_WIN_SIZE, 440, 706 MPI_WIN_UNIFIED, 441, 462, 481, 490, 706 MPI_WTIME_IS_GLOBAL, 361, 362, 381, 689, 703

## **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, 18, <u>18</u> , 19, 27, 87, <u>87</u> , 89, 92, 95,	MPI_T_pvar_handle, 611, <u>611</u> , 612–615, 709
103-105, 108-110, 120, 141, 142, 429,	MPI_T_pvar_session, 611, <u>611</u> , 612–615, 709
431, 433, 436, 437, 444, 446, 450, 452,	MPI_Win, 294–296, 305, 306, 370, 371, 379,
454-458,  460,  564,  567,  660,  689,  711	$\underline{429}, \underline{431}, \underline{433}, \underline{436}, 439, 441, 442, 444,$
MPI_Comm, 14, <u>26</u> , 250, 255–260, <u>263</u> , 264,	446, 450, 452, 454-458, 460, 466-476,
$267, 268, \underline{270}, 280-283, 288, 291-293,$	683,704,711
703, 704, 711	
MPI_Count, 19, <u>19</u> , 27, 711, 840	
MPI_Datatype, <u>87</u> , 669, 701–704, 711	
$MPI_ERR 374$	
MPI_Errhandler, <u>368</u> , 369–373, 683, 700, 704,	
711	
MPI_F08_status, <u>685</u> , 709, 711, 843	
MPI_File, 371, 372, 380, <u>519</u> , 521, 523–525,	
527, 530, 532, 536-547, 549-553,	
555-560, 564, 572, 573, 683, 704, 711	
MPI_Fint, 682, <u>682</u> , 709, 711, 847	
MPI_Group, 248, <u>248</u> , 249–255, 260, 281, 441,	
467, 469, 525, 682, 683, 704, 711	
MPI_Info, 363, 391, <u>391</u> , 392–395, 400, 403,	
406, 412, 414-418, 442, 519, 522, 527,	
530, 683, 703, 704, 711, 850	
MPI_Message, <u>70</u> , 683, 699, 704, 711, 841	
MPI_Offset, 18, <u>18</u> , 19, 27, 523, 524, 530, 532,	
536-540, 546, 547, 553, 555, 556, 568,	
575, <u>575</u> , 681, 711	
MPI_Op, 176, <u>185</u> , 188, 190–197, 212–217,	
229–234, 450, 452, 454, 458, 460, 683,	
704, 711	
MPI_Request, 51–53, <u>55</u> , 56, 57, 59–64, 66, 74,	
76-80, 502, 505, 538-540, 544-546,	
550, 662, 683, 704, 711	
MPI_Status, <u>30</u> , 32–34, 55, 56, 59–64, 66–68,	
70-72, 75, 81, 82, 115, 503, 508, 509,	
536, 537, 541-543, 549, 551, 552,	
556-560, 637, 684-686, 708, 711, 837,	
840, 843	
MPI_T_cvar_handle, 603, <u>603</u> , 604, 709	
MPI_T_enum, 598, <u>598</u> , 599, 600, 609, 709	
MPI_T_ERR_XXX, 622	

## 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.

15	MPI_Comm_copy_attr_function, 20, 21, 289, 636, 705, 712
16 17	MPI_Comm_delete_attr_function, 20, 289,
17	705, 712
	MPI_Comm_errhandler_fn, 628, 846
19	MPI_Comm_errhandler_function, 20, 368, 628,
20	630,712,846
21	MPI_Copy_function, 20, 625, 706, 717
22	MPI_Datarep_conversion_function, 568, 705,
23	712
24	MPI_Datarep_extent_function, 567, 712
25	MPI_Delete_function, 20, 626, 706, 717 MPI_File_errhandler_fn, 628, 846
26	MPI_File_errhandler_function, 371, 628, 712,
27	846
28	MPI_Grequest_cancel_function, 504, 712
29	MPI_Grequest_free_function, 503, 712
30	MPI_Grequest_query_function, 503, 712
31	MPI_Handler_function, 20, 630, 840
32	MPI_Type_copy_attr_function, 297, 705, 712
33	MPI_Type_delete_attr_function, 297, 705,
34	712, 844
35	MPI_User_function, 186, 189, 712
36	MPI_Win_copy_attr_function, 294, 705, 712 MPI_Win_delete_attr_function, 294, 705, 712
37	MPI_Win_delete_attr_function, 294, 705, 712 MPI_Win_errhandler_fn, 628, 846
38	MPI_Win_errhandler_function, 370, 628, 712,
39	846
40	
40	
42	
43	
43	
44 45	
46	
47	
48	

## **MPI Function Index**

The underlined page numbers refer to the function definitions.

MPI SET STATUS CANCELLED, 836 MPI_ABORT, 186, 366, 383, 386, 424, 597, 682, 850 MPI_ACCUMULATE, 12, 427, 443, 450, 451, 453, 459, 463, 487, 493, 494, 847, 850 MPI_ADD_ERROR_CLASS, 377, 377 MPI_ADD_ERROR_CODE, 378 MPI_ADD_ERROR_STRING, 378, 378, 379 MPI_ADDRESS, 20, 629, 645, 840 MPI_AINT_ADD, 21, 103-105, 105, 437, 838 MPI_AINT_DIFF, 21, 103-105, 105, 437, 838 MPI_ALLGATHER, 143, 147, 148, 167, 167, 168–170, 207 MPI_ALLGATHER_INIT, 224 MPI_ALLGATHERV, 143, 147, 148, 168, 169, 208MPI_ALLGATHERV_INIT, 225 MPI_ALLOC_MEM, 363, 364, 365, 375, 431-435, 438, 445, 474, 646-648, 660, 837, 844 MPI_ALLOC_MEM_CPTR, 364, 837 MPI_ALLREDUCE, 143, 146-148, 178, 185, 190, 190, 213, 848 MPI_ALLREDUCE_INIT, 230 MPI_ALLTOALL, 143, 147, 148, 170, 170, 171-173, 209, 845 MPI_ALLTOALL_INIT, 226 MPI_ALLTOALLV, 143, 147, 148, 172, 172, 173, 175, 210, 845 MPI_ALLTOALLV_INIT, 227 MPI_ALLTOALLW, 143, 147, 148, 174, 175, 212,845 MPI_ALLTOALLW_INIT, 228 MPI_ATTR_DELETE, 20, 300, 626, 627 MPI_ATTR_GET, 20, 300, 627, 689, 690 MPI_ATTR_PUT, 20, 300, 627, 689, 690, 692, 693 MPI_BARRIER, 143, 147, 149, 149, 150, 201, 484, 485, 577 MPI_BARRIER_INIT, 219 MPI_BCAST, 12, 143, 147, 150, 150, 151, 177, 202, 238

11 MPI_BCAST_INIT, 12, 219 MPI_BSEND, 12, 39, 40, 48 12MPI_BSEND_INIT, 77, 80 13 MPI_BUFFER_ATTACH, 46, 55 14MPI_BUFFER_DETACH, 47, 843 15MPI_CANCEL, 20, 43, 55, 66, 74, 74, 75, 200, 16218, 385, 456, 501, 504, 505, 628, 835 17MPI_CART_COORDS, 311, 325, 325, 849 18 MPI_CART_CREATE, 279, 310-312, 312, 313, 314, 324, 332-334, 661, 837, 848 19MPI_CART_GET, 311, 324, 324, 849 20MPI_CART_MAP, 311, 332, 333, 842 21MPI_CART_RANK, 311, 325, 325, 849 22MPI_CART_SHIFT, 311, 330, 330, 331, 334, 23849  24 MPI_CART_SUB, 311, 331, 332, 333, 849 25MPI_CARTDIM_GET, 311, 324, 324, 849 26MPI_CLOSE_PORT, 413, 413, 417 MPI_COMM_ACCEPT, 412, 414, 414, 415, 27422, 423 28 MPI_COMM_C2F, 682 29MPI_COMM_CALL_ERRHANDLER, 379, 30 38031MPI_COMM_COMPARE, 257, 280 32 MPI_COMM_CONNECT, 375, 415, 415, 422, 33 423 34 MPI_COMM_CREATE, 255, 257, 260, 261-266, 311, 846 35 MPI_COMM_CREATE_ERRHANDLER, 20, 36 367, <u>368</u>, 369, 629, 714, 716, 844 37 MPI_COMM_CREATE_GROUP, 257, 263, 38 <u>263</u>, 264, 265, 842 39 MPI_COMM_CREATE_KEYVAL, 20, 287, 40 288, 290, 300, 625, 688, 689, 713, 715, 41 844.848 MPI_COMM_DELETE_ATTR, 20, 287, 42290-292, 293, 300, 627 43 MPI_COMM_DISCONNECT, 300, 405, 423, 44424, 424 45MPI_COMM_DUP, 250, 255, 257, 258, 258, 46259, 261, 281, 283, 287, 289, 293, 300, 47307, 625, 835, 841 48

5 6

	874
1	MPI_COMM_DUP_FN, 20, 290, 290, 291,
2	641, 705, 838, 844
3	MPI_COMM_DUP_WITH_INFO, 257, 258
4	259, 260, 269, 841
5	MPI_COMM_F2C, <u>682</u>
6	MPI_COMM_FREE, 255, 258, <u>268</u> , 281, 28
7	290, 291, 293, 300, 384, 387, 405, 400, 400, 400, 400, 400, 400, 400
	424,626
8	MPI_COMM_FREE_KEYVAL, 20, 287, <u>29</u>
9	300, 626
10	MPI_COMM_GET_ATTR, 20, 287, 292, 29
11	300, 360, 627, 642, 689, 690, 692
12	MPI_COMM_GET_ERRHANDLER, 20, 3 369, 629, 849
13	<u>309</u> , 029, 849 MPI_COMM_GET_INFO, 269, 270, 270, 2
14	836, 841
15	MPI_COMM_GET_NAME, 303, <u>303</u> , 304,
16	MPI_COMM_GET_PARENT, 304, 367, 40
17	404, 404, 405
18	MPI_COMM_GROUP, 16, 248, 250, <u>250</u> , 2
19	256, 280, 367, 849
20	MPI_COMM_IDUP, 255, 257, 259, <u>259</u> , 260
21	278, 287, 289, 293, 300, 835, 837, 8
22	MPI_COMM_IDUP_WITH_INFO, 257, 26
23	$\frac{260}{269}, 269, 836$
	MPI_COMM_JOIN, 425, 425, 426
24	MPI_COMM_NULL_COPY_FN, 20, 290, 2
25	291, 636, 688, 705, 838, 844 MPI_COMM_NULL_DELETE_FN, 20, 29
26	<u>290</u> , 291, 705, 838
27	MPI_COMM_RANK, 256, <u>256</u> , 280, 643
28	MPI_COMM_RANK_F08, 643
29	MPI_COMM_REMOTE_GROUP, <u>281</u>
30	MPI_COMM_REMOTE_SIZE, 281, 281

- 31MPI_COMM_SET_ATTR, 20, 287, 290, 291, 32 300, 626, 642, 689, 690, 693
- MPI_COMM_SET_ERRHANDLER, 20, 367, 33 369, 629 34
- MPI_COMM_SET_INFO, 219, 269, 270, 270, 35 836, 837, 841 36
- MPI_COMM_SET_NAME, 302, 302, 303 37
- MPI_COMM_SIZE, 255, 256, 280
- 38MPI_COMM_SPAWN, 382, 389, 398-400, 400, 39401, 403-405, 407-409, 421-423
- MPI_COMM_SPAWN_MULTIPLE, 383, 389, 40398, 399, 404, 406, 407, 422, 423 41
- MPI_COMM_SPLIT, 257, 261, 264, 265, 266, 42307, 311, 312, 314, 332-334, 846
- 43MPI_COMM_SPLIT_TYPE, <u>267</u>, 269, 842
- 44MPI_COMM_TEST_INTER, 279, 280
- 45MPI_COMM_WORLD, 515

48

- 46MPI_COMPARE_AND_SWAP, 427, 443, 455, 47493
- MPI_CONVERSION_FN_NULL, 569, 705, 838 MPI_CWIN_GET_ATTR, 642 MPI_DIMS_CREATE, 311-313, 313, 836 MPI_DIST_GRAPH_CREATE, 269, 310, 311, 316, <u>318</u>, 319, 321, 329, 330, 334, 846 MPI_DIST_GRAPH_CREATE_ADJACENT, 269, 310, 311, 316, <u>316</u>, 317, 321, 329, 334, 842, 846 MPI_DIST_GRAPH_NEIGHBORS, 311, 328, 329, 329, 334, 842, 846 MPI_DIST_GRAPH_NEIGHBORS_COUNT, 311, 328, <u>328</u>, 329, 330, 839, 846 MPI_DUP_FN, 20, 290, 626, 706 MPI_ERRHANDLER_C2F, 683 MPI_ERRHANDLER_CREATE, 20, 629, 840, 844 MPI_ERRHANDLER_F2C, 683 MPI_ERRHANDLER_FREE, 367, 373, 384, 849 MPI_ERRHANDLER_GET, 20, 629, 840, 849 MPI_ERRHANDLER_SET, 20, 629, 840 MPI_ERROR_CLASS, 374, 374, 622 MPI_ERROR_STRING, 373, 373, 374, 377, 379 MPI_EXSCAN, 144, 147, 178, 185, <u>196</u>, 197, 217,846 MPI_EXSCAN_INIT, 234 MPI_F_SYNC_REG, 104, 495, 634, 650, 650, 651, 670, 672-675, 844 MPI_FETCH_AND_OP, 427, 443, 451, 453, 454MPI_FILE_C2F, 683 MPI_FILE_CALL_ERRHANDLER, 380, 380 MPI_FILE_CLOSE, 424, 519, 520, <u>521</u>, 522 MPI_FILE_CREATE_ERRHANDLER, 367, <u>371</u>, 372, 714, 716, 844 MPI_FILE_DELETE, 521, 522, 522, 526, 529, 581MPI_FILE_F2C, <u>683</u> MPI_FILE_GET_AMODE, 525, 525 MPI_FILE_GET_ATOMICITY, 573, 573 MPI_FILE_GET_BYTE_OFFSET, 541, 547, 547, 548, 554 MPI_FILE_GET_ERRHANDLER, 367, 372, 581, 849 MPI_FILE_GET_GROUP, 525, 525 MPI_FILE_GET_INFO, 526, 527, 528,
  - 529, 836, 850
  - MPI_FILE_GET_POSITION, 547, 547
- MPI_FILE_GET_POSITION_SHARED, 552, 553, 553, 573
- MPI_FILE_GET_SIZE, <u>524</u>, 525, 576

- 257, <u>258</u>,
- 281, 283,
  - 7, 405, 423,
- 287, 291,
- 292, 292, ), 692
- R, 20, 367,
- , <u>270</u>, 271,
- 3, 304, 848
- 367, 401,
- , <u>250</u>, 255,
- 259, 260,
- 5, 837, 841257, 260,
- 0, 290, 290,
- 20, 290,
- 643
- 281
- , <u>281</u>

## **Unofficial Draft for Comment Only**

MPI_FILE_GET_TYPE_EXTENT, 563, 564,
570
MPI_FILE_GET_VIEW, 532, <u>532</u>
MPI_FILE_IXXX, 534
MPI_FILE_IREAD, 533, 544, <u>544</u> , 554, 571
MPI_FILE_IREAD_ALL, 533, 545, <u>545</u> , 839
MPI_FILE_IREAD_AT, 533, 538, 538
MPI_FILE_IREAD_AT_ALL, 533, 539, 539,
839
MPI_FILE_IREAD_SHARED, 533, 550, 550
MPI_FILE_IWRITE, 533, <u>545</u> , 546
MPI_FILE_IWRITE_ALL, 533, 546, <u>546</u> , 839
MPI_FILE_IWRITE_AT, 533, <u>539</u> , 540
MPI_FILE_IWRITE_AT_ALL, 533, 540, <u>540</u> ,
$\frac{839}{8}$
MPI_FILE_IWRITE_SHARED, 533, 550, 551
MPI_FILE_OPEN, 376, 512, 519, <u>519</u> , 520,
521, 526, 529, 531, 548, 575, 576, 581,
582
$MPI_FILE_PREALLOCATE, 523, 524, \underline{524},$
571, 576
MPI_FILE_READ, 533, 541, <u>541</u> , 542, 544,
575, 576
MPI_FILE_READ_ALL, 533, 542, <u>542</u> , 545,
554, 555
MPI_FILE_READ_ALL_BEGIN, 533, 554,
555, <u>557</u> , 571, 675
MPI_FILE_READ_ALL_END, 533, 554, 555,
<u>558</u> , 571, 675
MPI_FILE_READ_ALL_FN, 12
MPI_FILE_READ_AT, 533, 536, <u>536</u> , 537, 538
MPI_FILE_READ_AT_ALL, 533, <u>536</u> , 537,
539
MPI_FILE_READ_AT_ALL_BEGIN, 533,
<u>555</u> , 675
MPI_FILE_READ_AT_ALL_END, 533, 556,
675 MDI EUE DEAD AT ALL EN 19
MPI_FILE_READ_AT_ALL_FN, 12
MPI_FILE_READ_ORDERED, 533, 551, 552
MPI_FILE_READ_ORDERED_BEGIN, 533,
<u>559,</u> 675
MPI_FILE_READ_ORDERED_END, 533,
559, 675
MPI_FILE_READ_ORDERED_FN, 12
MPI_FILE_READ_SHARED, 533, 549, <u>549</u> ,
550,552
MPI_FILE_SEEK, <u>546</u> , 547
MPI_FILE_SEEK_SHARED, 552, 553, <u>553</u> ,
554, 573
MPI_FILE_SET_ATOMICITY, 521, 572, <u>572</u>
MPI_FILE_SET_ERRHANDLER, 367, <u>372</u> ,
581
MPI_FILE_SET_INFO, 526, 527, <u>527</u> , 529,
836, 850
000,000

MPI_FILE_SET_SIZE, 523, 523, 524, 571,	1
574, 576	2
MPI_FILE_SET_VIEW, 98, 376, 520, 526,	3
527, 529, 530, <u>530</u> , 531, 532, 547, 554,	4
561, 567, 575, 582, 850, 851	
MPI_FILE_SYNC, 522, 533, 571–573, <u>573</u> , 579	5
MPI_FILE_WRITE, 533, 543, <u>543</u> , 544, 546,	6
575	7
MPI_FILE_WRITE_ALL, 533, 543, 544, 546	8
MPI_FILE_WRITE_ALL_BEGIN, 533, 558,	9
662, 675	10
MPI_FILE_WRITE_ALL_END, 533, 558, 675	11
MPI_FILE_WRITE_ALL_FN, 12	
MPI_FILE_WRITE_AT, 533, 537, 538,	12
540	13
MPI_FILE_WRITE_AT_ALL, 533, 537, 538,	14
540	15
MPI_FILE_WRITE_AT_ALL_BEGIN, 533,	16
556, 675	17
MPI_FILE_WRITE_AT_ALL_END, 533, 557,	18
675	19
MPI_FILE_WRITE_AT_ALL_FN, 12	20
MPI_FILE_WRITE_ORDERED, 533, 551,	
552, <u>552</u>	21
MPI_FILE_WRITE_ORDERED_BEGIN,	22
533, <u>560,</u> 675	23
MPI_FILE_WRITE_ORDERED_END, 533,	24
560, 675	25
MPI_FILE_WRITE_ORDERED_FN, 12	26
$MPI_FILE_WRITE_SHARED, 533, 534, \underline{549},$	27
550 - 552	28
MPI_FINALIZE, 17, 24, 360, 361, 383, <u>383</u> ,	
384 - 388, 424, 511, 520, 593, 603, 616,	29
618, 682, 684, 685, 842, 850	30
MPI_FINALIZED, 382, 385, 387, <u>387</u> , 388,	31
510, 516, 682, 838	32
MPI_FREE_MEM, 364, <u>364</u> , 365, 375, 432,	33
433	34
MPI_GATHER, 143, 146–148, <u>151</u> , 153, 154,	35
161, 162, 167, 177, 203	36
MPI_GATHER_INIT, <u>220</u>	
MPI_GATHERV, 143, 147, 148, 153, <u>153</u> , 154,	37
155, 163, 169, 204	38
MPI_GATHERV_INIT, <u>221</u>	39
MPI_GET, 12, 427, 443, <u>446</u> , 447, 453, 458,	40
463, 483, 486, 495, 673, 850	41
MPI_GET_ACCUMULATE, 427, 443, 451,	42
$\underline{452}, 453, 454, 461, 487, 493, 837$	43
MPI_GET_ADDRESS, 20, 87, 103, 104, <u>104</u> ,	44
105, 117, 437, 629, 645, 664, 669, 687,	45
688 MDI CDTT COLINIT, 22, 22, 24, 54, 116, 456	46
MPI_GET_COUNT, 33, <u>33</u> , 34, 54, 116, 456,	
509, 535, 841, 847	47
	48

1	MPI_GET_ELEMENTS, 54, 115, <u>115</u> , 116,
2	509, 510, 535, 841
3	MPI_GET_ELEMENTS_X, 54, 115, <u>115</u> , 116,
4	509, 535, 840
	MPI_GET_LIBRARY_VERSION, 360, 360,
5	382, 385, 510, 837, 838, 840
6	MPI_GET_PROCESSOR_NAME, 362, 362,
7	363, 849
8	MPI_GET_VERSION, <u>359</u> , 360, 382, 385, 510,
9	516, 649, 838
10	MPI_GRAPH_CREATE, 310, 311, 314, <u>314</u> ,
	316, 320, 323, 326, 327, 333, 334, 848,
11	849
12	MPI_GRAPH_GET, 311, 323, <u>323</u>
13	MPI_GRAPH_MAP, 311, <u>333</u> , 334
14	
15	MPI_GRAPH_NEIGHBORS, 311, 326, <u>326</u> ,
16	327, 334, 846
	MPI_GRAPH_NEIGHBORS_COUNT, 311,
17	326, <u>326</u> , 327, 846
18	MPI_GRAPHDIMS_GET, 311, 323, <u>323</u>
19	MPI_GREQUEST_COMPLETE, 502–505, 505
20	MPI_GREQUEST_START, <u>502</u> , 503, 714, 716,
21	847
22	$MPI_GROUP_C2F, \underline{683}$
	MPI_GROUP_COMPARE, <u>249</u> , 252
23	MPI_GROUP_DIFFERENCE, <u>251</u>
24	MPI_GROUP_EXCL, <u>252</u> , 253, 254
25	MPI_GROUP_F2C, <u>683</u>
26	MPI_GROUP_FREE, 255, <u>255</u> , 256, 367, 384,
27	849
28	MPI_GROUP_INCL, 252, <u>252</u> , 254
	MPI_GROUP_INTERSECTION, <u>251</u>
29	MPI_GROUP_RANGE_EXCL, 254, 254
30	MPI_GROUP_RANGE_INCL, <u>253</u> , 254
31	MPI_GROUP_RANK, <u>248</u> , 256
32	MPI_GROUP_SIZE, <u>248</u> , 256
33	MPI_GROUP_TRANSLATE_RANKS, 249,
34	249,848
35	MPI_GROUP_UNION, <u>250</u>
	MPI_IALLGATHER, 143, 147, 148, 207
36	MPI_IALLGATHERV, 143, 147, 148, 208
37	MPI_IALLREDUCE, 143, 147, 148, 213
38	MPI_IALLTOALL, 143, 147, 148, 209
39	MPI_IALLTOALLV, 143, 147, 148, <u>210</u>
40	MPI_IALLTOALLW, 143, 147, 148, <u>211</u>
41	MPI_IBARRIER, 143, 147, 199, 201, <u>201</u> , 238
	MPI_IBCAST, 12, 143, 147, 202, <u>202</u> , 242
42	MPI_IBSEND, <u>51</u> , 55, 80
43	MPI_IEXSCAN, 144, 147, <u>217</u>
44	MPI_IGATHER, 143, 147, 148, <u>203</u>
45	MPI_IGATHERV, 143, 147, 148, <u>205</u> MPI_IGATHERV, 143, 147, 148, <u>204</u>
46	MPI_IGATHERV, 143, 147, 148, <u>204</u> MPI_IMPROBE, 12, 66, 69, 70, <u>70</u> , 71, 72, 74,
47	
	512, 837, 841 MDI IMPECV 60, 71, 72, 72, 74, 841
48	MPI_IMRECV, 69, 71–73, <u>73</u> , 74, 841

MPI_INEIGHBOR_ALLGATHER, 311, 344, 842 MPI_INEIGHBOR_ALLGATHERV, 311, 345, 842 MPI_INEIGHBOR_ALLTOALL, 311, 346, 842 MPI_INEIGHBOR_ALLTOALLV, 311, 347, 842 MPI_INEIGHBOR_ALLTOALLW, 312, 348, 842 MPI_INFO_C2F, 683 MPI_INFO_CREATE, 392, 392 MPI_INFO_DELETE, 375, 393, 393, 395 MPI_INFO_DUP, 395, 395 MPI_INFO_F2C, 683 MPI_INFO_FREE, 271, 384, 395, 443, 528 MPI_INFO_GET, 391, 393, 850 MPI_INFO_GET_NKEYS, 391, 394, 394, 395, 850 MPI_INFO_GET_NTHKEY, 391, 395, 850 MPI_INFO_GET_VALUELEN, 391, 394, 850 MPI_INFO_SET, 392, 392, 393, 395 MPI_INIT, 17, 24, 247, 360, 361, 381, 381, 382, 385-388, 401-403, 405, 421, 422, 513, 514, 516, 593, 596, 603, 616, 681, 682, 684, 685, 841, 842, 845, 846 MPI_INIT_THREAD, 247, 382, 387, 513, 514-516, 593, 596, 603, 681, 841, 842, 846 MPI_INITIALIZED, 382, 385, 386, 386, 387, 388, 510, 516, 682, 838 MPI_INTERCOMM_CREATE, 257, 264, 281, 282, 282, 283, 842 MPI_INTERCOMM_MERGE, 257, 264, 278, 281-283, <u>283</u>, 844 MPI_IPROBE, 12, 33, 66, 67, <u>67</u>, 68–72, 74, 512, 841 MPI_IRECV, 12, <u>53</u>, 73, 663, 664, 667, 668 MPI_IREDUCE, 143, 147, 148, 212, 213 MPI_IREDUCE_SCATTER, 143, 147, 148, 215MPI_IREDUCE_SCATTER_BLOCK, 143, 147, 148, 214 MPI_IRSEND, 53 MPI_IS_THREAD_MAIN, 510, 514, 516, 838 MPI_ISCAN, 144, 147, 216 MPI_ISCATTER, 143, 147, 148, 205 MPI_ISCATTERV, 143, 147, 148, 206 MPI_ISEND, 12, 51, 80, 641, 642, 645, 662, 663, 668, 669 MPI_ISSEND, 52 MPI_KEYVAL_CREATE, 20, 625, 627, 717 MPI_KEYVAL_FREE, 20, 300, 626

MPI_LOOKUP_NAME, 375, 412, 416, 418, 418

MPI_MESSAGE_C2F, <u>683</u>, 841 MPI_MESSAGE_F2C, <u>683</u>, 841 MPI_MPROBE, 12, 66, 69-71, 71, 72, 74, 512, 841 MPI_MRECV, 12, 69, 71, 72, 72, 73, 841 MPI_NEIGHBOR_ALLGATHER, 311, 335, 337, 338, 344, 842 MPI_NEIGHBOR_ALLGATHER_INIT, 349 MPI_NEIGHBOR_ALLGATHERV, 311, 337, 345, 842 MPI_NEIGHBOR_ALLGATHERV_INIT, 350 MPI_NEIGHBOR_ALLTOALL, 311, 339, 340, 346, 842 MPI_NEIGHBOR_ALLTOALL_INIT, 351 MPI_NEIGHBOR_ALLTOALLV, 311, 340, 347.842 MPI_NEIGHBOR_ALLTOALLV_INIT, 352 MPI_NEIGHBOR_ALLTOALLW, 311, 341, 342, 349, 842 MPI_NEIGHBOR_ALLTOALLW_INIT, 353 MPI_NULL_COPY_FN, 20, 21, 290, 626, 706 MPI_NULL_DELETE_FN, 20, 290, 626, 706 MPI_OP_C2F, 683 MPI_OP_COMMUTATIVE, 192, 846 MPI_OP_CREATE, 185, <u>185</u>, 187, 641, 712, 715, 843, 844 MPI_OP_F2C, 683 MPI_OP_FREE, 188, 384 MPI_OPEN_PORT, 412, <u>412</u>, 414, 415, 417, 418 MPI_PACK, 49, <u>134</u>, 137, 140, 565, 568 MPI_PACK_EXTERNAL, 8, 140, 141, 655, 848 MPI_PACK_EXTERNAL_SIZE, 142 MPI_PACK_SIZE, 48, 137, <u>137</u>, 841 MPI_PCONTROL, 588, 589, 589 MPI_PROBE, 12, 31, 33, 34, 66-68, 68, 69, 70, 72, 74, 512, 841 MPI_PUBLISH_NAME, 412, 416, 416, 417, 418 MPI_PUT, 12, 427, 443, 444, 446, 450, 451, 457, 463, 468, 478, 480, 484, 486, 495, 662, 673, 850 MPI_QUERY_THREAD, 510, 515, 516, 838 MPI_RACCUMULATE, 427, 443, 451, 453, 458, 459 MPI_RECV, 12, 26, 30, 32–34, 67, 69, 71, 86, 114, 115, 135, 144, 152, 239, 510, 577, 616, 669, 672, 673 MPI_RECV_INIT, 12, 79, 79 MPI_REDUCE, 143, 147, 148, 176, <u>176</u>, 177, 178, 185–187, 190, 193, 195–197, 212, 213, 450, 451, 453, 454, 847 MPI_REDUCE_INIT, 229

MPI_REDUCE_LOCAL, 177, 178, 185, <u>191</u> ,	1
844, 846	2
MPI_REDUCE_SCATTER, 143, 147, 148,	3
$178, 185, 194, \underline{194}, 195, 215$	4
MPI_REDUCE_SCATTER_BLOCK, 143,	5
147, 148, 178, 185, 193, 193, 194, 214, 846	6
MPI_REDUCE_SCATTER_BLOCK_INIT,	7
231	8
MPI_REDUCE_SCATTER_INIT, $\underline{232}$	9
MPI_REGISTER_DATAREP, 376, <u>565</u> ,	10
567-569, 582, 714, 717	11
MPI_REQUEST_C2F, <u>683</u>	12
MPI_REQUEST_F2C, 683	13
MPI_REQUEST_FREE, 57, 57, 74, 80, 200,	14
218, 383, 384, 456, 504, 505, 845	15
MPI_REQUEST_GET_STATUS, 34, 66, <u>66</u> ,	16
503, 845	
MPI_RGET, 427, 443, <u>457</u> , 458	17
MPI_RGET_ACCUMULATE, 427, 443, 451,	18
453, <u>460</u> , 461 MPI_RPUT, 427, 443, <u>456</u> , 457	19
$MPI_RSEND, 12, 39, 41$	20
MPI_RSEND_INIT, 78	21
MPI_SCAN, 144, 147, 178, 185, <u>195</u> , 196, 197,	22
216	23
MPI_SCAN_INIT, <u>233</u>	24
MPI_SCATTER, 143, 147, 148, 161, <u>161</u> , 163,	25
164, 193, 205	26
MPI_SCATTER_INIT, <u>222</u>	27
MPI_SCATTERV, 143, 147, 148, 163, <u>163</u> ,	
164, 195, 206	28
MPI_SCATTERV_INIT, <u>223</u>	29
MPI_SEND, 12, 25, <u>26</u> , 27, 34, 36, 86, 113,	30
114, 134, 239, 520, 577, 590, 669, 670,	31
672,673	32
MPI_SEND_INIT, 12, <u>76</u> , 80	33
MPI_SENDRECV, <u>81</u> , 330	34
MPI_SENDRECV_REPLACE, $\underline{82}$	35
MPI_SIZEOF, 634, <u>656</u> , 657	36
MPI_SSEND, <u>41</u>	37
MPI_SSEND_INIT, 77	38
MPI_START, 79, <u>79</u> , 80, 218, 669	39
MPI_STARTALL, 80, <u>80</u> , 218, 669	
MPI_STATUS_C2F, <u>684</u> MDL_STATUS_C2F08_685_842	40
MPI_STATUS_C2F08, <u>685</u> , 843 MPI_STATUS_F082C, <u>685</u> , 843	41
MPI_STATUS_F082C, <u>085</u> , <u>845</u> MPI_STATUS_F082F, <u>686</u> , <u>843</u>	42
MPI_STATUS_F082F, <u>080</u> , 843 MPI_STATUS_F2C, <u>684</u>	43
MPI_STATUS_F2F08, <u>686</u> , 843	44
MPI_STATUS_SET_CANCELLED, <u>509</u>	45
MPI_STATUS_SET_ELEMENTS, <u>508</u> , 509	46
MPI_STATUS_SET_ELEMENTS_X, 509,	47
<u>509</u> , 840	48

1 MPI_T_CATEGORY_CHANGED, 622  $\mathbf{2}$ MPI_T_CATEGORY_GET_CATEGORIES, 621, <u>621</u>, 622, 624 3 MPI_T_CATEGORY_GET_CVARS, 621, 621, 4 622.624  $\mathbf{5}$ MPI_T_CATEGORY_GET_INDEX, 620, 620, 6 624, 839  $\overline{7}$ MPI_T_CATEGORY_GET_INFO, 619, 620, 8 622, 624, 838, 839 9 MPI_T_CATEGORY_GET_NUM, 619 MPI_T_CATEGORY_GET_PVARS, 621, 621, 10 622, 624 11MPI_T_CVAR_GET_INDEX, 602, 602, 624, 12839 13MPI_T_CVAR_GET_INFO, 598, 600, 600, 14601, 603-605, 622, 624, 838, 839 15MPI_T_CVAR_GET_NUM, 600, 604 16MPI_T_CVAR_HANDLE_ALLOC, 598, 603, 17604, 605, 624 18MPI_T_CVAR_HANDLE_FREE, 604, 604, 624 19MPI_T_CVAR_READ, 604 20MPI_T_CVAR_WRITE, 604 21MPI_T_ENUM_GET_INFO, 598, 598, 624 22MPI_T_ENUM_GET_ITEM, 598, 599, 624 23MPI_T_FINALIZE, 597, 597  24 MPI_T_INIT_THREAD, 596, 596, 597 25MPI_T_PVAR_GET_INDEX, 610, 610, 624, 839 26MPI_T_PVAR_GET_INFO, 598, 608, 609, 27609, 612, 614, 615, 622, 624, 838, 839 28MPI_T_PVAR_GET_NUM, 608, 612 29MPI_T_PVAR_HANDLE_ALLOC, 598, 612, 30 612, 614, 624  31 MPI_T_PVAR_HANDLE_FREE, <u>612</u>, 613, 32 624,838 MPI_T_PVAR_READ, 614, 615, 624, 838 33 MPI_T_PVAR_READRESET, 610, 615, 615, 34624.838 35 MPI_T_PVAR_RESET, 615, 615, 624, 838 36 MPI_T_PVAR_SESSION_CREATE, 611, 624 37 MPI_T_PVAR_SESSION_FREE, 611, 624 38MPI_T_PVAR_START, 613, 624, 838 39 MPI_T_PVAR_STOP, 613, 624, 838 MPI_T_PVAR_WRITE, 614, 614, 624, 838 40MPI_TEST, 34, 54, 56, <u>56</u>, 57, 59, 61, 74, 80, 41 218, 383, 505, 534, 535 42MPI_TEST_CANCELLED, 54-56, 75, 75, 503, 43510, 535 44MPI_TESTALL, 59, 62, 62, 503-505, 508, 511 45MPI_TESTANY, 59, 60, 61, 64, 503-505, 508, 46511 47MPI_TESTSOME, 59, 64, 64, 65, 503-505, 508, 511 48

MPI_TOPO_TEST, 311, 322, 322 MPI_TYPE_C2F, 682 MPI_TYPE_COMMIT, 112, 112, 683 MPI_TYPE_CONTIGUOUS, 14, 87, 87, 89, 107, 119, 518, 563 MPI_TYPE_CREATE_DARRAY, 14, 33, 99, 99, 119 MPI_TYPE_CREATE_F90_COMPLEX, 13, 119, 121, 179, 565, 634, 653, 655 MPI_TYPE_CREATE_F90_INTEGER, 13, 119, 121, 178, 565, 634, 653, 655 MPI_TYPE_CREATE_F90_REAL, 13, 119, 121, 179, 565, 634, 652, 653-655, 845 MPI_TYPE_CREATE_HINDEXED, 14, 20, 87, 92, <u>92</u>, 94, 96, 119, 629 MPI_TYPE_CREATE_HINDEXED_BLOCK, 14, 87, 94, 94, 119, 841 MPI_TYPE_CREATE_HVECTOR, 14, 20, 87, 89, 89, 119, 629 MPI_TYPE_CREATE_INDEXED_BLOCK, 14, 93, 94, 119 MPI_TYPE_CREATE_KEYVAL, 287, 297, 300, 689, 713, 716, 848 MPI_TYPE_CREATE_RESIZED, 20, 87, 107, <u>109</u>, 110, 119, 563, 630, 843 MPI_TYPE_CREATE_STRUCT, 14, 20, 87, 94, 95, 95, 96, 107, 119, 175, 629 MPI_TYPE_CREATE_SUBARRAY, 14, 16, <u>96, 98, 100, 119</u> MPI_TYPE_DELETE_ATTR, 287, 299, 300, 843 MPI_TYPE_DUP, 14, 113, 113, 119, 843 MPI_TYPE_DUP_FN, 297, 297, 705, 838 MPI_TYPE_EXTENT, 20, 629, 840 MPI_TYPE_F2C, 682 MPI_TYPE_FREE, 112, 121, 298, 384 MPI_TYPE_FREE_KEYVAL, 287, 298, 300 MPI_TYPE_GET_ATTR, 287, 299, 300, 642, 689.843 MPI_TYPE_GET_CONTENTS, 118, 119, 120, 121, 122 MPI_TYPE_GET_ENVELOPE, 118, 118, 120, 121, 654 MPI_TYPE_GET_EXTENT, 20, 108, 111, 629, 657, 687 MPI_TYPE_GET_EXTENT_X, 109, 840 MPI_TYPE_GET_NAME, 305, 844 MPI_TYPE_GET_TRUE_EXTENT, 110, 110 MPI_TYPE_GET_TRUE_EXTENT_X, 110, 111, 840 MPI_TYPE_HINDEXED, 20, 629, 840 MPI_TYPE_HVECTOR, 20, 629, 840

MPI_TYPE_INDEXED, 14, 90, 91, <u>91</u>, 92, 93, 119

MPI_TYPE_LB, 20, 629, 840	MPI_WIN_FLUSH, 434, 456, 457, <u>474</u> , 475,
MPI_TYPE_MATCH_SIZE, 634, 657, 657, 844	479,  493,  494
MPI_TYPE_NULL_COPY_FN, 297, 297, 705,	MPI_WIN_FLUSH_ALL, 456, 457, <u>475</u> , 479
838	MPI_WIN_FLUSH_LOCAL, 456, <u>475</u> , 479
MPI_TYPE_NULL_DELETE_FN, <u>297</u> , 705,	MPI_WIN_FLUSH_LOCAL_ALL, 456, <u>475</u> ,
838, 844	476, 479
MPI_TYPE_SET_ATTR, 287, <u>299</u> , 300, 642,	MPI_WIN_FREE, 295, 384, 424, 439, <u>439</u> , 440
689, 693, 843	MPI_WIN_FREE_KEYVAL, 287, <u>295</u> , 300
MPI_TYPE_SET_NAME, <u>305</u> , 843	MPI_WIN_GET_ATTR, 287, 296, 300, 440,
MPI_TYPE_SIZE, 106, <u>106</u> , <u>590</u> , 841	689, 693
MPI_TYPE_SIZE_X, 106, 106, 840	MPI_WIN_GET_ERRHANDLER, 367, <u>371</u> ,
MPI_TYPE_STRUCT, 20, 629, 840	849
MPI_TYPE_UB, 20, 629, 840	MPI_WIN_GET_GROUP, 441, 441
MPI_TYPE_VECTOR, 14, 88, <u>88</u> , 89, 92, 119	MPI_WIN_GET_INFO, 441, 442, 442, 443,
MPI_UNPACK, 135, 135, 136, 140, 568	836, 841
MPI_UNPACK_EXTERNAL, 8, <u>141</u> , 655	MPI_WIN_GET_NAME, <u>306</u>
MPI_UNPUBLISH_NAME, 375, 417, 417	MPI_WIN_LOCK, 430, 439, 464, 471,
MPI_WAIT, 32, 34, 54, 55, <u>55</u> , 56–58, 60, 61,	472-474, 476, 477, 479, 483-485
74, 80, 199, 218, 239, 383, 501, 505,	MPI_WIN_LOCK_ALL, 430, 464, 472, <u>472</u> ,
512, 534, 535, 554, 571, 572, 662, 668,	473, 476, 477, 479, 485, 493
669, 672	MPI_WIN_NULL_COPY_FN, 294, 294, 705,
MPI_WAITALL, 59, 61, <u>61</u> , 62, 200, 240, 456,	838
503–505, 508, 511, 512	MPI_WIN_NULL_DELETE_FN, 294, 705,
MPI_WAITANY, 43, 59, 59, 60, 64, 503–505,	838
508, 511, 512	MPI_WIN_POST, 439, 463, 468, 469, <u>469</u> ,
MPI_WAITSOME, 59, 63, <u>63</u> , 64, 65, 503–505,	470, 471, 473, 476, 477, 479, 486, 488
508, 511, 512 MDI WIN ALLOCATE 428, 421, 422, 424	MPI_WIN_SET_ATTR, 287, <u>295</u> , 300, 440,
MPI_WIN_ALLOCATE, 428, <u>431</u> , 432, 434,	642, 689, 693
439, 441, 445, 474, 646, 648, 837	MPI_WIN_SET_ERRHANDLER, 367, <u>370</u>
MPI_WIN_ALLOCATE_CPTR, 432, 837	MPI_WIN_SET_INFO, 441, 442, 442, 836, 841
MPI_WIN_ALLOCATE_SHARED, 428, 433,	MPI_WIN_SET_NAME, <u>305</u>
<u>433</u> , 435, 439, 441, 648, 837	MPI_WIN_SHARED_QUERY, 433, <u>435</u> , 648,
MPI_WIN_ALLOCATE_SHARED_CPTR,	837
434, 837	MPI_WIN_SHARED_QUERY_CPTR, 436,
MPI_WIN_ATTACH, 436, 437, <u>437</u> , 438, 439,	837
474	MPI_WIN_START, 439, 463, <u>467</u> , 468, 469,
MPI_WIN_C2F, <u>683</u>	471, 476, 477, 486, 493
MPI_WIN_CALL_ERRHANDLER, <u>379</u> , 380	MPI_WIN_SYNC, 476, <u>476</u> , 479, 481, 482,
MPI_WIN_COMPLETE, 439, 463, 468, <u>468</u> ,	485, 493–495
469-471, 479, 485	MPI_WIN_TEST, $470, \underline{470}$
MPI_WIN_CREATE, 428, <u>429</u> , 431, 432, 434,	MPI_WIN_UNLOCK, 439, 457, 464, <u>472</u> , 474,
437 - 439,  441,  478,  512	479,  483,  484
MPI_WIN_CREATE_DYNAMIC, 376, 428,	MPI_WIN_UNLOCK_ALL, 457, 464, 472,
$436, \underline{436}, 437 - 439, 441, 478$	$\underline{473}, 479, 483, 493$
MPI_WIN_CREATE_ERRHANDLER, 367,	MPI_WIN_WAIT, 439, 463, 469, <u>469</u> , 470,
$\underline{370}, 371, 714, 716, 844$	471, 473, 479, 483, 485, 486
MPI_WIN_CREATE_KEYVAL, 287, <u>293</u> , 300,	MPI_WTICK, 21, 381, <u>381</u>
689,713,715,848	MPI_WTIME, 21, 362, <u>380</u> , 381, 590, 608
MPI_WIN_DELETE_ATTR, 287, <u>296</u> , 300	mpiexec, 382, 387, 388, <u>388</u> , 514
MPI_WIN_DETACH, 436, 438, <u>438</u> , 439	mpirun, 388
MPI_WIN_DUP_FN, 294, <u>294</u> , 705, 838	
MPI_WIN_F2C, <u>683</u>	PMPI_, 587, 642
MPI_WIN_FENCE, 439, 447, 463, <u>466</u> , 467,	PMPI_AINT_ADD, 21
476, 477, 479, 480, 483, 488, 673	PMPI_AINT_DIFF, 21

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

 $\mathbf{6}$ 

 $\overline{7}$ 

8

9

15

17

 24 

40

45

46

47

1	PMPI_ISEND, 642, 6	345
---	--------------------	-----

- 2 PMPI_WTICK, 21
- PMPI_WTIME, 21
- $\mathbf{5}$

- $\overline{7}$

- 21
- 24

- 28

- 41